

# X/O/P/E/N

PORTABILITY GUIDE

SYSTEM V SPECIFICATION  
COMMANDS & UTILITIES

X/O/P/E/N

PORTABILITY GUIDE  
SYSTEM V SPECIFICATION COMMANDS & UTILITIES

1



# X/O/P/E/N/

PORTABILITY GUIDE

SYSTEM V SPECIFICATION  
COMMANDS & UTILITIES



© 1987, The X/OPEN Group Members

*All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.*

#### X/OPEN PORTABILITY GUIDE

Set of 5 Volumes

ISBN: 0-444-70179-6

Volume 1	XVS Commands and Utilities	ISBN: 0-444-70174-5
Volume 2	XVS System Calls and Libraries	ISBN: 0-444-70175-3
Volume 3	XVS Supplementary Definitions	ISBN: 0-444-70176-1
Volume 4	Programming Languages	ISBN: 0-444-70177-X
Volume 5	Data Management	ISBN: 0-444-70178-8

*Published by:*

ELSEVIER SCIENCE PUBLISHERS B.V.  
P.O Box 1991  
1000 BZ Amsterdam  
The Netherlands

*Sole distributors for the U.S.A. and Canada:*


ELSEVIER SCIENCE PUBLISHING COMPANY, INC.  
52 Vanderbilt Avenue  
New York, N.Y. 10017  
U.S.A.

*Any comments relating to the material contained in the X/OPEN Portability Guide may be submitted to the X/OPEN Group by letter via the Publisher or directly by Electronic Mail to:*

*xopen@inset.co.uk*

PRINTED IN THE NETHERLANDS





# **Contents**

## **PREFACE**

## **THE COMMON APPLICATIONS ENVIRONMENT**

1. The Common Applications Environment
2. System V
3. Internationalisation
4. C Language
5. Other Programming Languages
6. Data Management
7. Source Code Transfer Between Machines
8. Networking And Communications

## **XVS COMMANDS AND UTILITIES**

1. Introduction
2. Commands And Utilities

## **XVS COMMANDS AND UTILITIES APPENDIX A**

1. Enhanced Definitions





## **Trademarks**

UNIX<sup>™</sup> is a registered trademark of AT&T in the USA and other countries.

C-ISAM<sup>™</sup> is a trademark of Informix Corporation.

LEVEL II COBOL<sup>™</sup> is a trademark of Micro Focus Limited.

XENIX<sup>™</sup> is a trademark of Microsoft Inc.

IBM<sup>™</sup> is a trademark of International Business Machines Corp.

X/OPEN<sup>™</sup> is a licensed trademark of the X/OPEN Group Members.

POSIX<sup>™</sup> is a trademark of the Institute of Electrical and Electronic Engineers Inc.





## Preface

X/OPEN represents a major breakthrough in the world of standards for the information technology industry. Ten\* of the world's major information system suppliers have come together to encourage applications portability resulting in tangible benefits for computer users, independent software houses and for the suppliers themselves.

The Group's principal aim is to increase the volume of applications available and to maximise the return on investments in software development made by Users and Independent Software Vendors.

This is achieved by ensuring portability of application programs at the source code level. Through this portability, users can mix and match computer systems and applications software from many suppliers, and thus investment in applications software is protected into the future.

In order to provide such portability, the Group defines a **Common Applications Environment** built on the interfaces to the **UNIX** operating system, as defined in the AT&T System V Interface Definition, and covering other aspects required of a comprehensive applications interface.

The X/OPEN Portability Guide contains an evolving portfolio of practical standards for application portability. All of the members of X/OPEN guarantee to support the standards defined, leading to:

- Growing portability
- No dependence on a single source - freedom of choice
- Increased application software selection
- More security in software investments
- International support for the *Common Applications Environment*

X/OPEN is not a standards-setting organisation; it is a joint initiative by members of the business community to integrate evolving standards into a common, beneficial and continuing strategy.

---

\* At the time of publication, the membership of the X/OPEN Group was BULL, DEC, ERICSSON, HEWLETT-PACKARD, ICL, NIXDORF, OLIVETTI, PHILIPS, SIEMENS and UNISYS



Issue 2 of the X/OPEN Portability Guide (published in January 1987) comprises five volumes defining the interfaces currently identified as components of the Common Applications Environment.

Volume 1	System V Specification: Commands and Utilities
Volume 2	System V Specification: System Calls and Libraries
Volume 3	System V Specification: Supplementary Definitions XVS Internationalisation XVS Terminal Interfaces XVS Inter-Process Communication XVS Source Code Transfer
Volume 4	Programming Languages C Language COBOL Language
Volume 5	Data Management Indexed Sequential Access Method (ISAM) Relational Database Language (SQL)

In addition, each volume includes an introduction giving the philosophy of the Common Applications Environment and an overview of its components.

This guide is aimed at both the decision makers and the implementation teams of:

- Independent Software Vendors
- Software Houses
- Users
- Equipment Manufacturers

The Guide is designed to sit permanently on the desk, serving as a common reference point for anyone directly concerned with the practical side of software development, namely systems designers, programmers and consultants.

The various parts of the Portability Guide are closely interrelated. Any reference from one part of the definition to another part uses the title of the other part as a reference (e.g., "XVS TERMINAL INTERFACES").



## **Acknowledgements**

X/OPEN gratefully acknowledges:

- **AT&T** for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The **/usr/group** Standards Committee, whose Standard contributed to the Group's work.
- **Informix Corporation**, of Menlo Park, California (Telex no. 361834) for permission to use material from the specification of their C-ISAM product and for provision of that material in machine readable form.
- **Micro Focus Ltd.** of Newbury, Berkshire for permission to use material from the specification of their LEVEL II COBOL compiler.
- The assistance given by the following companies in the preparation of the Database Language (SQL) definition:

Informix Corporation  
Oracle Corporation  
Queensland Information Technology  
Relational Technology Inc.  
Unify Corporation



## **Referenced Documents**

The following documents are referenced in this guide:

- System V Interface Definition (Spring 1985 - Issue 1)
- System V Interface Definition (Spring 1986 - Issue 2)
- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2)
- UNIX System V - Release 2.0 Programming Guide (April 1984 - Issue 2)
- ANS Draft Proposal for C Language (October 1986 - ANS X3J11/86-151)
- 1984 /usr/group Standard
- IEEE P1003.1 Trial Use Standard (April 1986)
- Informix Corporation C-ISAM Reference Manual (Version 2.10 - January 1985)
- MicroFocus Level II COBOL Language Reference Manual (Version 2.5 and 2.6, Issue 7 - April 1984)
- Standard for COBOL (ANS X3.23-1974)
- Standard for COBOL (ANS X3.23-1985)
- Standard for FORTRAN (ANS X3.9-1978)
- Standard for Database Language (SQL) (ANS X3.135-1986)
- Standard for PASCAL (ISO 7185-1983)

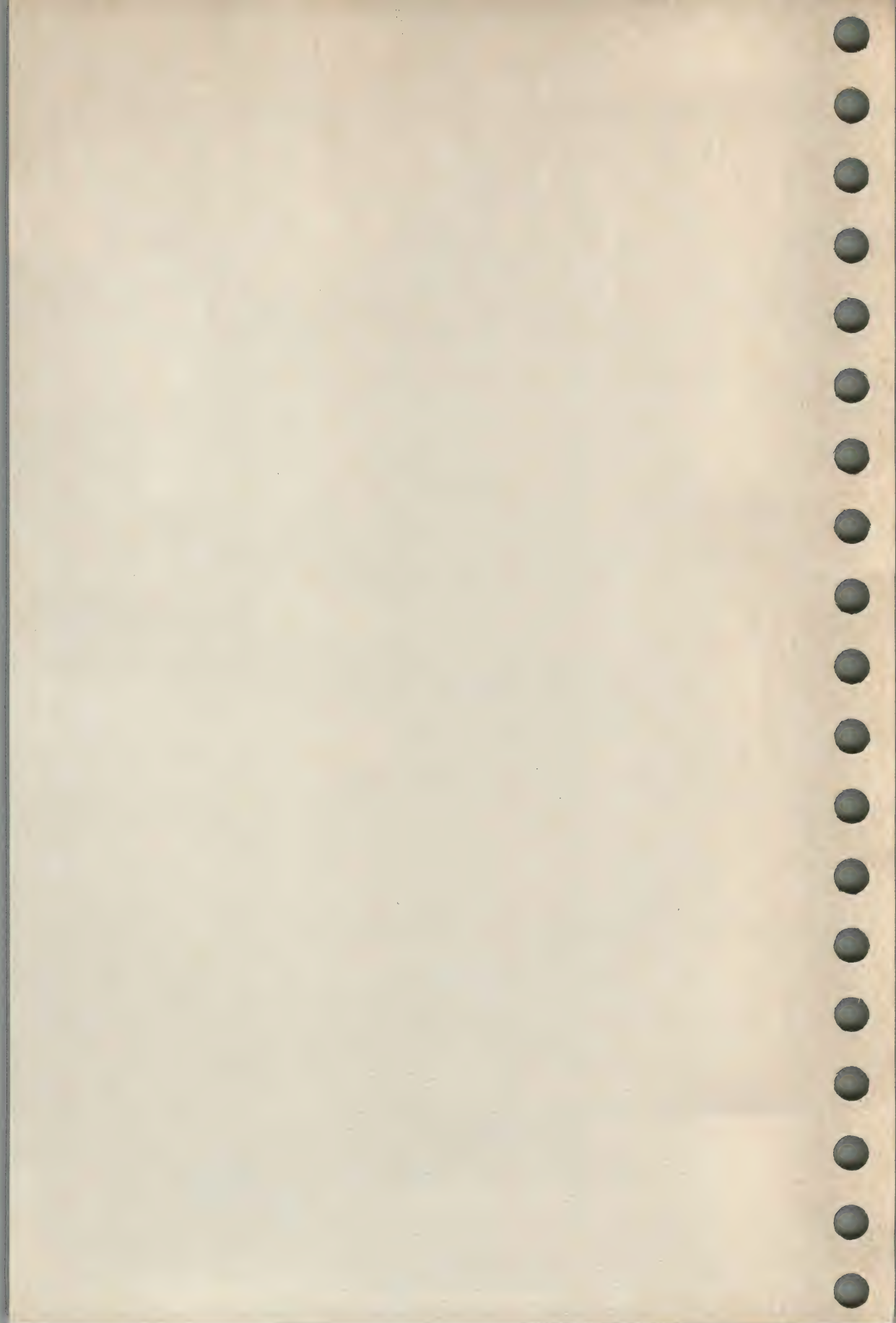


# X/O/P/E/N/

PORTABILITY GUIDE

THE COMMON APPLICATIONS  
ENVIRONMENT







# **Contents**

Chapter	1	THE COMMON APPLICATIONS ENVIRONMENT
Chapter	2	SYSTEM V
	2.1	INTRODUCTION
	2.2	THE EVOLVING STANDARD
	2.2.1	Origins
	2.2.2	The IEEE "POSIX" Standard
	2.2.3	The AT&T System V Interface Definition
	2.3	THE X/OPEN SYSTEM V SPECIFICATION
	2.3.1	System Calls and Libraries
	2.3.2	Inter-process Communication
	2.3.3	Commands and Utilities
Chapter	3	INTERNATIONALISATION
	3.1	INTRODUCTION
	3.2	The X/OPEN NATIVE LANGUAGE SYSTEM
Chapter	4	C LANGUAGE
	4.1	INTRODUCTION
	4.2	C LANGUAGE PORTABILITY GUIDELINES
	4.3	THE ANS X3J11 DRAFT STANDARD
	4.4	THE C PROGRAM PORTABILITY CHECKER ( <i>lint</i> )
Chapter	5	OTHER PROGRAMMING LANGUAGES
	5.1	INTRODUCTION
	5.2	COBOL
	5.3	FORTRAN
	5.4	PASCAL
Chapter	6	DATA MANAGEMENT
	6.1	INTRODUCTION
	6.2	INDEXED SEQUENTIAL ACCESS METHOD (ISAM)
	6.3	RELATIONAL DATABASE LANGUAGE (SQL)



<b>Chapter</b>	<b>7</b>	<b>SOURCE CODE TRANSFER BETWEEN MACHINES</b>
	7.1	INTRODUCTION
	7.2	FLOPPY DISC STANDARD
	7.3	MAGNETIC TAPE
	7.4	UTILITIES
<b>Chapter</b>	<b>8</b>	<b>NETWORKING AND COMMUNICATIONS</b>
	8.1	NETWORKING AND COMMUNICATION
	8.2	OPEN SYSTEMS INTERCONNECTION
	8.3	GENERALISED INTER-PROCESS COMMUNICATION, IPC
	8.4	DISTRIBUTED FILE SYSTEM
	8.5	DISTRIBUTED TRANSACTION PROCESSING



# **The Common Applications Environment**

The formation of the X/OPEN Group represents a major initiative by an international group of suppliers of computer systems to create a free and open market, offering Independent Software Vendors (ISVs) as wide a market as possible for their products and giving users an increased return on investment in application software.

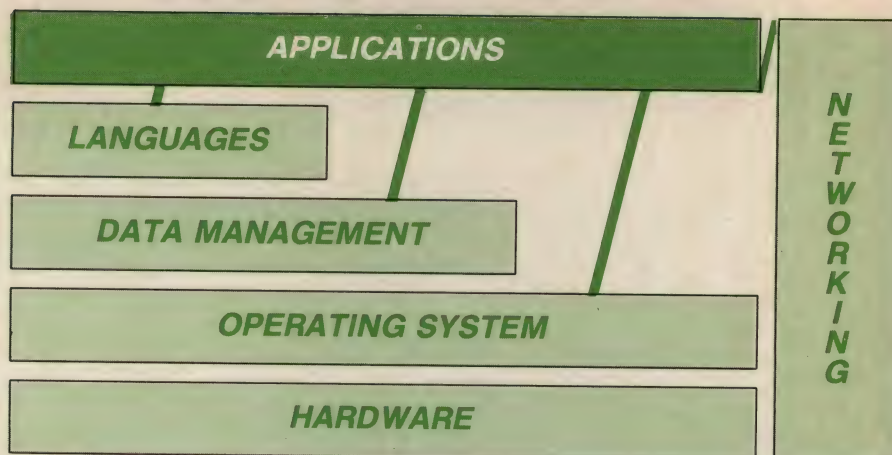
The current dominance of proprietary machine environments is restricting the growth of the computer industry. Users tend to get locked into a particular proprietary system by the investment they have made in the applications. Independent Software Vendors are discouraged from writing applications for a particular environment because of the limited markets caused by this fragmentation. This means that there is very little generally available software for each type of system, thus increasing the size of investment needed by each user. All this in turn limits the sales potential of machines from the computer suppliers.

The objective shared by the members of the X/OPEN Group is to establish a Common Applications Environment to the mutual advantage of users, Independent Software Vendors and computer suppliers. Applications written to operate in this environment will be portable at the source code level to a wide range of machines, thereby releasing the user from dependence on a single supplier, reducing the necessary investment in applications, considerably increasing the market for independent software and opening up the market for systems suppliers.

The existence of these "Open Systems" allows users to mix and match systems from different suppliers, and to move applications between machines to meet changing requirements as business grows, thereby giving protection of investment in applications software into the future.

The great increase in the potential market encourages the Independent Software Vendors to produce a wealth of general applications packages, and the availability of this further reduces the investment needed by the users. The whole situation is thus mutually reinforcing.





The foundations of the Common Applications Environment are the interfaces of the UNIX System V operating system, as defined in the AT&T "System V Interface Definition", and the C language.

To define a complete environment for portable applications, it is also necessary to satisfy the requirements for data management, integration of applications, data communications, distributed systems, the use of high level languages and the many other aspects involved in providing a comprehensive applications interface. The X/OPEN Group intends, therefore, to publish progressively definitions covering these areas.

The systems of the X/OPEN Group members that support interfaces derived from UNIX operating systems will do this according to the X/OPEN definitions and will support the full Common Applications Environment.

A specific Common Applications Environment feature may not, however, be present if it is not relevant in the market area in which a particular system is sold. For example, a system sold only in a scientific context might not support COBOL. Conversely, a particular system may support features over and above those of the Common Applications Environment, some of which may partially overlap. An example of this could be that an alternative dialect of COBOL is supported in addition to that of the Common Applications Environment.

The X/OPEN Group is primarily concerned with standards selection and adoption. The general policy is to use International Standards, where they exist, and to adopt "de facto" standards in other cases.

Where International Standards do not exist, it is X/OPEN policy to work closely with standardisation bodies to encourage their emergence.

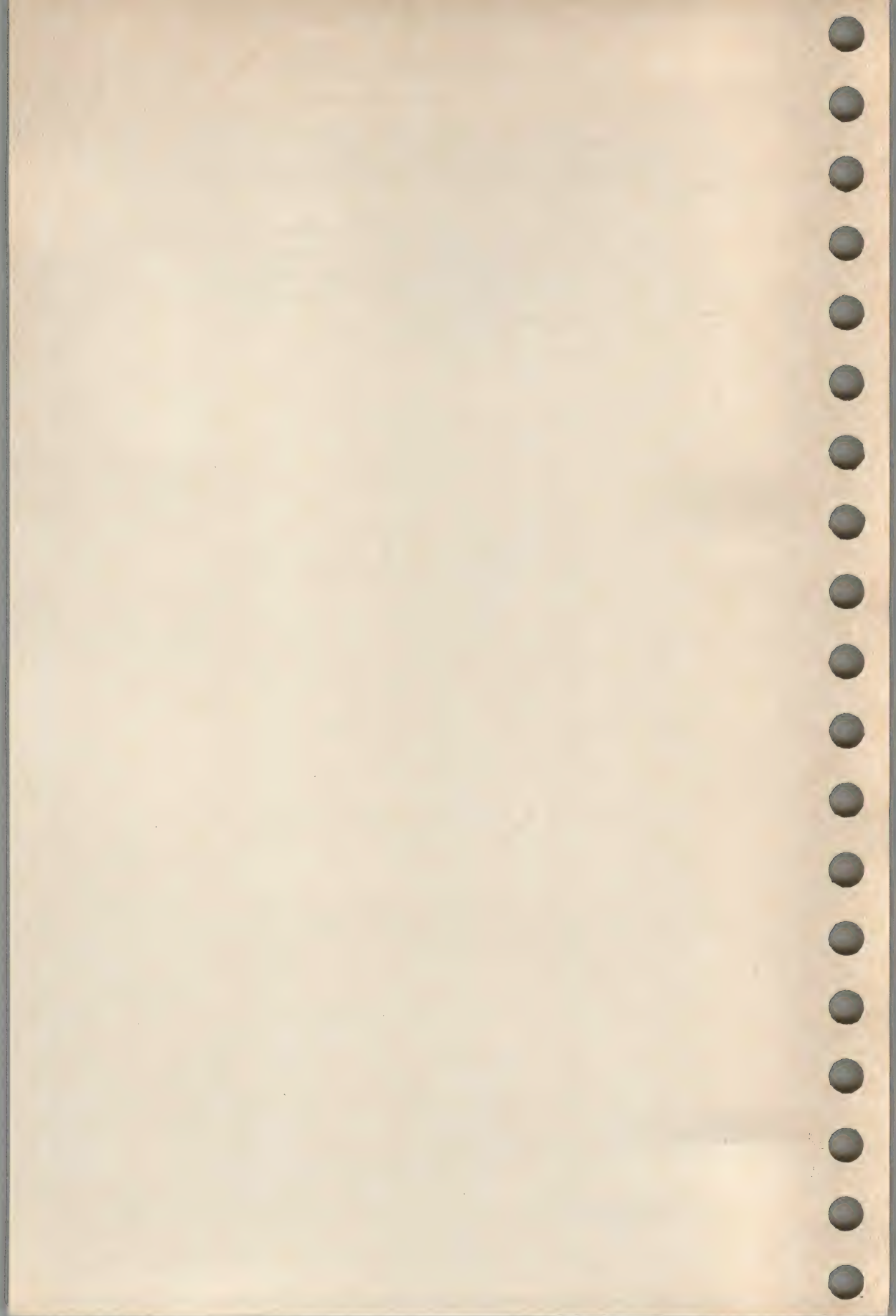


## *The Common Applications Environment*

It is important that the defined elements of the Common Applications Environment be readily achievable on member systems, and have wide acceptance. For this reason, the definitions, in general, fall within the capabilities of at least one currently available popular product.

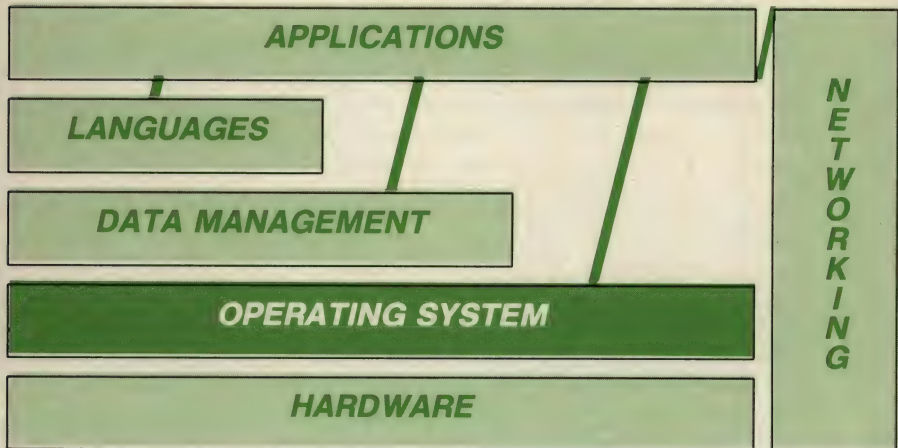
In this guide, certain aspects of the Common Applications Environment are defined with reference to the interfaces offered by specific products. This does not mean that member systems will necessarily contain these products, but that the defined interfaces will be supported. Indeed the method of support for an interface on a particular system may change with time.







# **X/OPEN System V Specification (XVS)**



## **2.1 INTRODUCTION**

The X/OPEN System V specification (XVS) defines the applications interfaces provided by the underlying operating system and forms the foundation of the Common Applications Environment.

The XVS is derived from a series of standards activities. The evolving standard is briefly addressed, and the relationship between the X/OPEN System V Specification and the System V Interface Definition and other standards is explained.



## 2.2 THE EVOLVING STANDARD

### 2.2.1 Origins

The UNIX operating system was developed by Ritchie and Thompson at Bell Laboratories in the early 1970s. The current AT&T System V version may be traced back directly to that first system.

For many years, it remained basically an academic product. More recently, computer suppliers have adopted the UNIX system as a multi-tasking, multi-user and portable operating environment. They have based their systems on one of several releases, variants or look-alikes. Of these, the most widely used were Version 7, System III, the Berkeley system and XENIX.

Although these systems had much in common, the degree of compatibility at the application interface level was insufficient to permit the development of totally portable applications.

### 2.2.2 The IEEE "POSIX" Standard

/usr/group, a group of users of UNIX derivatives in the USA, established a committee with the objective of proposing a set of standards for application level interfaces. After publishing its standard, together with a reviewer's guide, the group decided to seek IEEE status for the standard. In late 1984, the /usr/group standards committee closed its activities in its own name and its members were encouraged to become involved in the IEEE group, known as P1003.

The P1003 group published a "trial-use" standard in early 1986, which has the status of a "Draft American National Standard". This "Portable Operating System for Computer Environments" (POSIX) is expected to be revised and submitted for approval in 1987.

The IEEE P1003 group is working to extend the POSIX standard. It is expected that the next area to be standardised will be the subset of commands which offer an interface to applications.



### 2.2.3 The AT&T System V Interface Definition

The "System V Interface Definition" (SVID), first published by AT&T in the spring of 1985, represented a major standards initiative. AT&T were prominent in the activities of /usr/group and the influence of /usr/group can clearly be seen in the SVID. The stated purpose of the SVID is to define common interfaces for all System V implementations.

The definition groups interfaces into a mandatory *base* plus a series of *extensions*. The *base* interfaces must be present in any implementations of System V. If any interface from an *extension* is supported, it must adhere to the definition.

Issue 1 of the SVID comprised a single volume defining operating system interfaces (known as *system calls* and *library routines*) available to applications as directly called external functions and defined in terms of invocation from C-language programs.

Issue 2 of the SVID was published in early 1986 and comprised two volumes. The first volume contained the same material as issue 1, with some restructuring to improve ease of use and some changes to correct errors. It comprised the *Base System Definition* plus a single extension referred to as the *Kernel Extension*.

The second volume primarily defined commands and utilities, normally invoked through a command interpreter. It comprised further extensions referred to as the *Base Utilities Extension*, *Advanced Utilities Extension*, *Administered Systems Extension*, *Software Development Extension*, and *Terminal Interface Extension*. The latter two include library routines in addition to utilities.



## 2.3 THE X/OPEN SYSTEM V SPECIFICATION

The X/OPEN System V Specification (XVS) is based upon the AT&T System V Interface Definition, but also taking into account the trial use standard published by IEEE and the capabilities of the existing AT&T System V product.

The XVS is organised into a number of self-contained sections:

"XVS SYSTEM CALLS AND LIBRARIES" defines the Operating System Interfaces and broadly corresponds to Volume 1 of the SVID.

"XVS COMMANDS AND UTILITIES" defines commands and utilities and broadly corresponds to Volume 2 of the SVID. The purpose of the X/OPEN Portability Guide is to facilitate the portability of applications. As such, system administration is outside of its scope and the routines included in the AT&T *Administered System Extension* are not defined.

"XVS TERMINAL INTERFACES" defines a set of portable interfaces to locally connected asynchronous terminals and broadly corresponds to the AT&T Terminal Interface Extension in Volume 2 of the SVID.

"XVS INTER-PROCESS COMMUNICATION" defines interfaces to shared memory, semaphores and message passing, included as an interim mechanism to satisfy the immediate requirements of Inter-Process Communication facilities.

### 2.3.1 System Calls and Libraries

"XVS SYSTEMS CALLS AND LIBRARIES" contains a full definition of interfaces to system calls and library routines and broadly corresponds to Volume 1 of the SVID.

The X/OPEN Group has extended the SVID in a number of areas:

- Certain changes have been included, which the SVID denotes as future directions.
- The use of symbolic names to replace numeric constants, introduced by AT&T in their SVID, has been extended.
- Clarification of existing wording has been introduced in a limited number of places to "tighten" the specification.
- The opportunity has been taken to correct clerical errors in the SVID.
- Definitions have been included of a number of further UNIX System V Release 2.0 functions which are in widespread use by application developers.

The relationship between the XVS and the SVID is clearly stated. The whole of the SVID *base* definition is included as mandatory with the exception of the maths group, which is not mandatory for systems sold into markets where it is not relevant. *Termio* was not mandatory in issue 1 of the XVS because of some difficulties in implementation. These have now been resolved, and the routine is now mandatory, except in systems which do not support locally connected asynchronous lines.



The XVS incorporates all the interfaces within the SVID *kernel extension set* although a number are defined as optional.

In the XVS, interfaces defined as *optional* will be available on most but not necessarily all X/OPEN systems; use of them could restrict portability. Any *optional* interface supported on an X/OPEN system will conform to the X/OPEN specification.

The XVS defines interfaces in terms of their interface syntax and run-time behaviour, without constraining the method of their implementation. The names "system calls" and "subroutines" are retained purely for compatibility with other documentation.

### 2.3.2 Inter-process Communication

The kernel extension interfaces relating to shared memory, semaphores and message passing are included in "XVS INTER-PROCESS COMMUNICATION" as a short-term mechanism to satisfy the immediate requirements for Inter-Process Communication facilities. However, they are machine specific and cannot be supported on all hardware architectures. The Group believes that a more generalised approach to the whole subject of Inter-Process Communication is required.

### 2.3.3 Commands and Utilities

"XVS COMMANDS AND UTILITIES" contains a full definition of interfaces to commands and utilities and broadly corresponds to Volume 2 of the SVID. The interfaces are split functionally into those which are intended to provide an applications interface (referred to as *Standard Utilities*) and those which are only intended to be used by development programmers or during the porting of applications to an X/OPEN system (referred to as *Development Utilities*). The Standard Utilities will be present in all X/OPEN systems, as they are needed to provide a run-time interface to applications. The Development Utilities may only be present in development systems; their respective descriptions are clearly annotated to indicate this. This same distinction is also present in the SVID. The X/OPEN development utilities correspond exactly to the SVID *Software Development Extension*.

The definition of standards for commands and utilities is an evolving process and X/OPEN intends to participate fully.

The current definition is a valuable first step, but additional work will be needed to evolve towards a complete standard. For example:

- The number of options defined for many of the commands is excessive and includes functionality which is rarely used or is implementation specific.
- There are too many different ways of achieving the same results.
- Many of the current descriptions were written to record the observed behaviour of already existing utilities and the level of precision is inadequate for use as a definitive standard.



Rectification of this is an enormous task and the current X/OPEN definition is of necessity based on the available documentation, but incorporates extensive annotation to highlight potential portability problems resulting from the points listed above. To achieve maximum portability, application developers should avoid the use of the functionality so annotated.

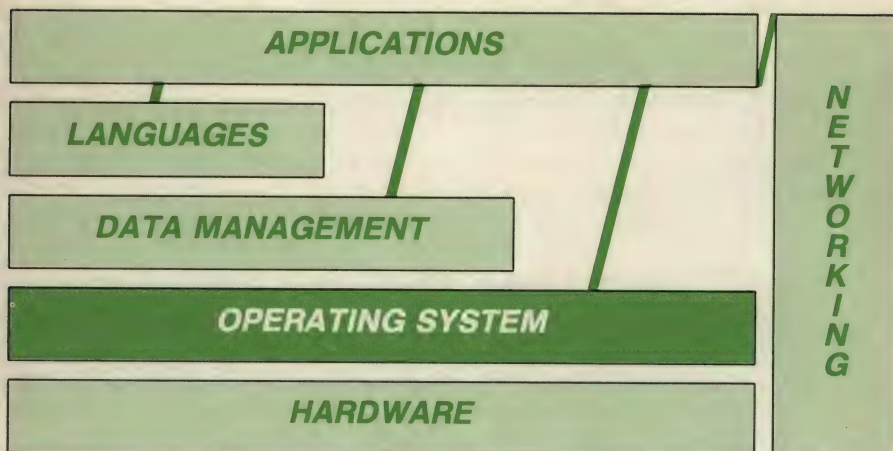
X/OPEN is working on a substantially improved definition of commands, with the number of options reduced to those in common use and with a higher level of specification. In addition to the current X/OPEN specification, "XVS COMMANDS AND UTILITIES" contains a proposed template for improved specifications together with a number of examples.

This improvement process will be carried out in close consultation with the various user organisations and standards bodies, such as IEEE and ISO, to ensure that the result is a single standard definition of operating system commands and utilities.

Readers of the X/OPEN Portability Guide are invited to participate directly in the consultative process to ensure that the evolving standard matches the requirements of existing and future applications.



## Internationalisation



### 3.1 INTRODUCTION

X/OPEN members market systems in many countries. Our customers and users speak many different languages and conform to different cultural conventions and business practices. It is important therefore that X/OPEN systems are capable of supporting a range of language and cultural environments. In many cases a strong requirement also exists to cope with these variations on the same system. An example is within the administration of the European Economic Community.

To date UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set and on American English. There are no facilities for dealing with other coded character sets, nor for supporting different languages and cultural conventions.

The requirement for effective mixed language working brings with it the need for coded character sets larger than can be accommodated by 7-bit characters, as does the requirement to support the more complex languages. At the same time there is a trade-off between the ability to handle larger coded character sets, and the amount of storage required to hold the data. For most European requirements an 8-bit system provides the correct balance. For the major Eastern languages (such as Chinese and Japanese) a 16-bit system is necessary, even to support a single language.



To satisfy these requirements enhancements must be made to the system to provide full data transparency to applications, allowing flexibility in the choice of coded character set(s) employed. Additionally, the system must allow program messages (both input and output) to be handled in the native language of each user, as well as providing cultural dependent data items (such as date formats and currency symbols).

### 3.2 THE X/OPEN NATIVE LANGUAGE SYSTEM

The X/OPEN Native Language System (NLS) is a set of interfaces designed to facilitate the development of applications that can operate in many different language and cultural environments. The interfaces have been derived from those of the Native Language Support system developed by the Hewlett-Packard Company of Palo Alto, California. They have been further enhanced by X/OPEN and have been modified in strategic areas to more closely relate to the Internationalisation proposals of the Draft Proposed American National Standard for the C Programming Language.

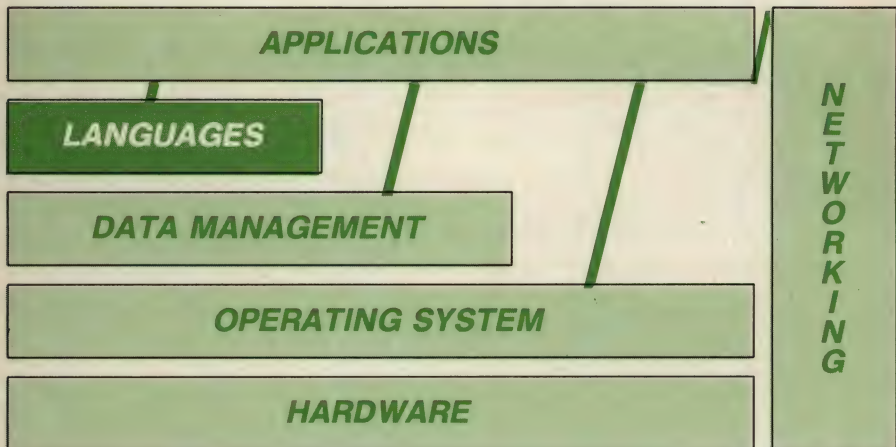
The first issue of the specification, defined in "XVS INTERNATIONALISATION", concentrates on facilities for the development of internationalised applications (rather than on internationalising the operating system itself), and on the 8-bit coded character set situations.

The following groups of facilities are defined:

- A message catalogue system which allows program messages to be held apart from the program logic, translated into different native languages, and the appropriate version retrieved by the program at run time.
- An announcement mechanism whereby native language, local custom (territory) and codeset requirements appropriate to each user can be identified to applications at run time.
- Enhanced interface definitions of standard C library functions, which provide language dependent character type classification, upper to lower case and lower to upper case character conversions, date and time messages, floating point to string conversions, and text collation.
- Library functions which allow programs to determine cultural and language specific data dynamically (e.g. the format of date and time strings, weekday and month names, currency symbols, etc.).
- A set of standard commands and library functions which will operate correctly with 8-bit characters.



# C Language



## 4.1 INTRODUCTION

This chapter addresses the C language and guidelines for portability when writing C code.

Currently the American National C Language Standards committee, X3J11, is working towards a standard for the C programming language. The X/OPEN Group is represented on that committee by member companies and intends to adopt the standard, once it has been established as a practical reality.

Meanwhile, the X/OPEN definition included in "C LANGUAGE" is based upon that given in Chapter 2 of the "System V Programming Guide", Release 2.0, published by AT&T.

## 4.2 C LANGUAGE PORTABILITY GUIDELINES

Whilst the C language provides the basis for applications portability, it is easy to write statements, using valid C constructs, that are machine specific. Care has to be taken when writing programs that are intended to be portable across a range of systems. "C LANGUAGE" includes advice towards ensuring portability.



#### 4.3 THE ANS X3J11 DRAFT STANDARD

The ANS X3J11 standard has been published in a draft form but may still change before it is approved. However, it is already clear that the standard will impose certain restrictions such that programs written to the current C language definition may not work correctly, if the source is later passed through a compiler that supports the ANS standard.

To address this, "C LANGUAGE" includes advice on writing programs to avoid these problems.

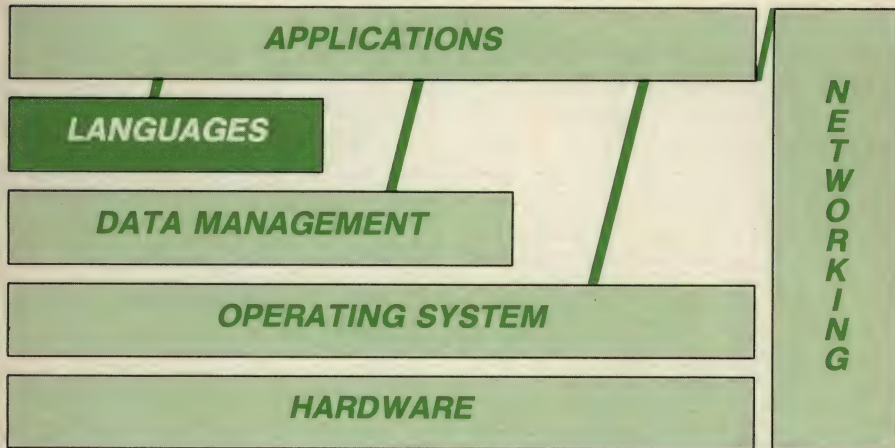
#### 4.4 THE C PROGRAM PORTABILITY CHECKER (*lint*)

The *lint* program checks C source programs for violation of most of the portability rules. It also gives a more stringent enforcement of the type rules of C than is provided by most C compilers. A further option detects a number of wasteful or error-prone constructions which nevertheless are syntactically correct.

Use of the *lint* program is recommended: it is described in "C LANGUAGE".



## Other Programming Languages



### 5.1 INTRODUCTION

This chapter addresses programming languages other than C on X/OPEN systems. It covers the inclusion of the principal high level languages in the Common Applications Environment.

To date, The X/OPEN Group has established definitions for COBOL and FORTRAN and PASCAL.



## 5.2 COBOL

The X/OPEN COBOL definition identifies a common set of language facilities that will be supported by COBOL compilers on all member systems. Applications written to this definition will be portable to any X/OPEN system.

The ISO Working Group and ANS COBOL committee have been working for some years towards a revised standard for COBOL to reflect more accurately the capabilities of modern COBOL compiling systems. The latest international standard, "X3.23 - 1985" was approved during 1985. At the time of publication of Issue 2 of the Portability Guide, there are few compilers in compliance with the revised standard and hence the X/OPEN group feels that major changes in the X/OPEN definition to reflect the new standard would be premature. It is likely, however, that any future edition of the X/OPEN COBOL language definition will relate to "COBOL 1985" and the new standard has been used as a reference when eliminating obsolete elements from the COBOL definition.

The most widely followed standard for COBOL is still that defined in the earlier 1974 Standard, "ANS X3.23-1974", to which most current COBOL compilers substantially conform.

The 1974 standard is incomplete in the area of facilities for interaction with the on-line user. To overcome this deficiency, most COBOL compilers provide extensions to the *ACCEPT* and *DISPLAY* verbs, but they do this in incompatible ways. Since the majority of applications now include interactive operation, it is necessary for a standard form of *ACCEPT* and *DISPLAY* to be defined in the X/OPEN Common Applications Environment.

In order to have an X/OPEN definition that is achievable on member systems within a short timescale, and one that would have immediate widespread acceptance, it has been based on the definition of COBOL embodied in a popular product: Micro Focus LEVEL II COBOL, which itself conforms to the "ANS X3.23-1974".

The Micro Focus LEVEL II COBOL language specification includes a number of other extensions beyond the 1974 standard, in addition to those to *ACCEPT* and *DISPLAY*. None of these are currently included in the X/OPEN definition. The X/OPEN definition also applies restrictions to the ANS-based parts of the LEVEL II definition.

Whilst the X/OPEN COBOL definition is based on the specification of a particular product, the means of implementation across the systems of the X/OPEN members may vary. Any particular system may support extensions beyond the facilities identified, but their use is likely to impede portability.

The X/OPEN COBOL definition is given in detail in "COBOL LANGUAGE" and its relationship to the 1974 standard is clearly shown.

The definition is given in terms of the command syntax derived from the LEVEL II COBOL Reference Manual. The semantics of the *ACCEPT* and *DISPLAY* verbs are defined in "COBOL LANGUAGE". The semantics of all other elements of the language are defined by the "X3.23-1974" standard.



### 5.3 FORTRAN

The X/OPEN definition for FORTRAN is the formal definition given in the American National Standards document "FORTRAN 77, ANS X3.9 - 1978". This has had wide-scale acceptance throughout the world and there are many certified compilers available.

The majority of FORTRAN compilers, while adhering to the basic FORTRAN 77 standard, also offer extensions beyond that standard. There is little compatibility in these extensions between compilers and they do not form part of the X/OPEN definition. Developers are warned that use of these extensions will affect the portability of FORTRAN programs.

### 5.4 PASCAL

The current X/OPEN definition for PASCAL is the formal definition given in the International Organisation for Standardisation document "Programming Languages - PASCAL" ISO 7185-1983 (level 1).

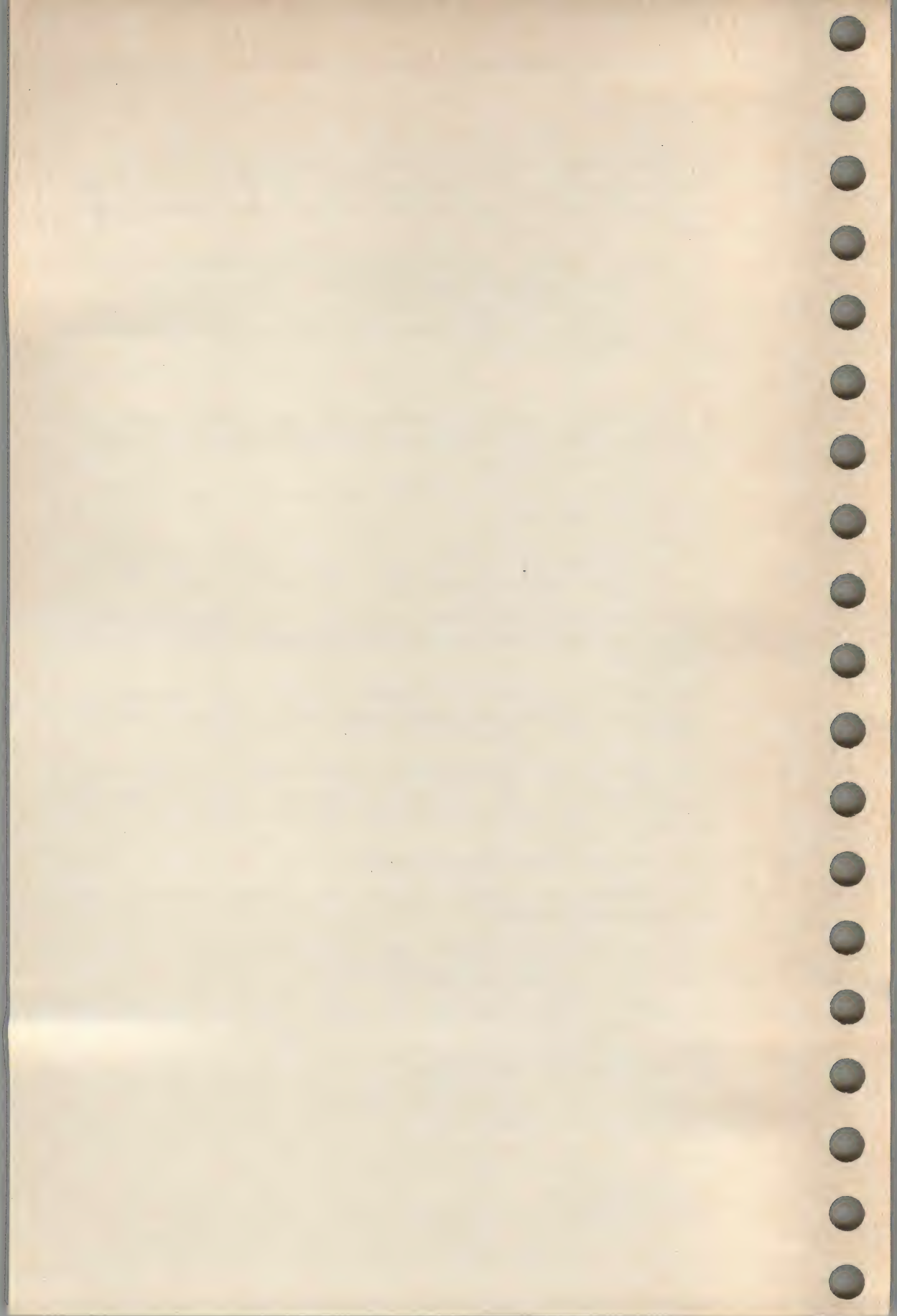
This is well accepted throughout the world and there are significant numbers of certified compilers available.

In order to enhance the portability of programs among PASCAL implementations on X/OPEN systems, the X/OPEN Group has decided to give a uniform definition for certain features designated as implementation-defined in ISO 7185.

- When the required identifiers "input" and "output" occur as program parameters they shall be bound by default to the external system files STDIN and STDOUT respectively.
- There shall be no implementation dependent restrictions on the base-type of a set-type which disallow 0 as the minimal ordinal value and 255 as the maximum ordinal value of that base-type.
- (lazy input) the underlying data transfer action required for a call to the predefined procedure "get" for a textfile (both explicit and where implied by "reset" and "read"), shall be postponed until one of the following events, if any, whichever occurs first :
  - a. access to the buffer-variable of the file
  - b. the buffer-variable of the file is passed as an actual variable parameter to a procedure or function
  - c. a call to the predefined function "eoln" for that file
  - d. a call to the predefined function "eof" for that file

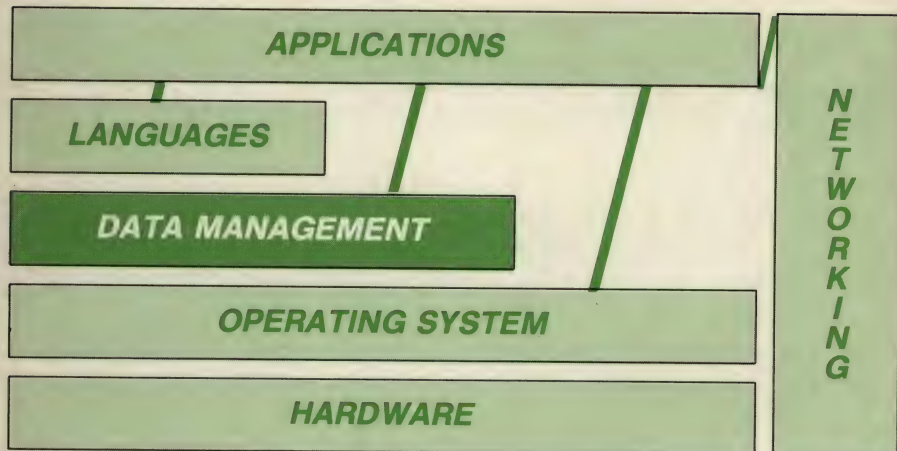
The majority of Pascal compilers also offer extensions beyond the current ISO Standard PASCAL definition. There is little compatibility in these extensions between compilers and developers are warned that use of these extensions may affect the portability of PASCAL programs.







## Data Management



### 6.1

#### INTRODUCTION

The input/output facilities supported by System V consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data Management is a key element in the integration of applications. Applications written in a variety of languages must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

Addressing these issues, the X/OPEN Group defines interfaces for the creation, management and manipulation of indexed files, generally known as the Indexed Sequential Access Method (ISAM) and for access to relational database management systems, the standard Relational Database Language (SQL).

The availability of these interfaces on X/OPEN systems will not only provide application portability, but will ease and encourage integration.



## 6.2 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

The X/OPEN definition for ISAM, which is contained in "INDEXED SEQUENTIAL ACCESS METHOD", is a major subset of the specification of the C-ISAM product, Version 2.10, published by Informix Corporation.

The full specification of C-ISAM contains implementation details specific to that product, in addition to the definition of the interface available to applications. Only the applications interface forms part of the X/OPEN definition; implementation details specific to C-ISAM have been omitted. Indeed, there are alternative implementations available on particular member systems.

## 6.3 RELATIONAL DATABASE LANGUAGE (SQL)

To reflect the growing significance of Relational Data Base systems, the X/OPEN group has defined application interfaces embedded within high level "host" languages to a relational database management system for a free-standing database.

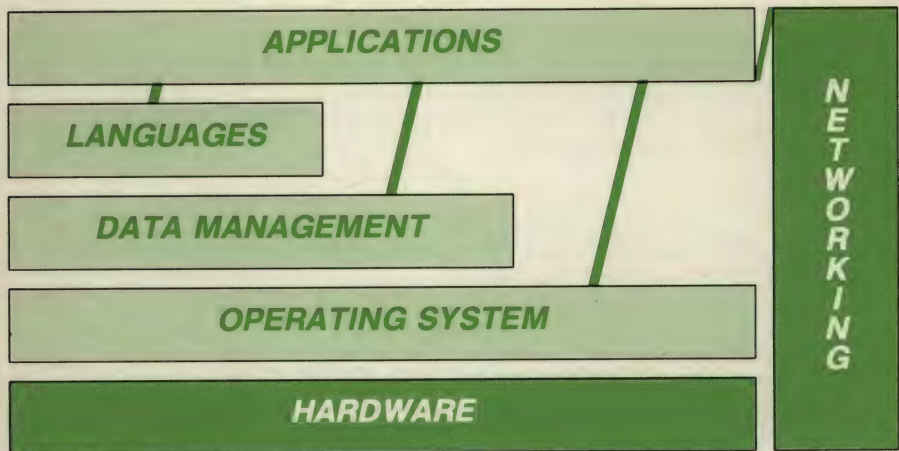
The widely accepted standard for access to relational data base is that defined in the American National Standard document Relational Database Language (SQL) "ANS X3.135-1986".

The X/OPEN definition is based closely on "X3.135-1986" but taking careful account of the capabilities of the leading relational database management systems currently available. The X/OPEN group has worked closely with the vendors of these products throughout.

"X3.135-1986" allows for two levels of compliance, Level 1 and Level 2. Most existing products comply only at Level 1 although it is expected that a significant number of products will have achieved full compliance with Level 2 before the end of 1987. "X3.135-1986" Level 1 SQL is not an adequate definition for application developers, since it leaves too many areas as implementor-defined. In preparing its definition, the X/OPEN group has examined these areas carefully and an agreed X/OPEN approach defined.

The X/OPEN SQL definition is contained in "RELATIONAL DATABASE LANGUAGE (SQL)", and contains a full description of the syntax and semantics of SQL together with a detailed comparison between the X/OPEN definition and the "ANS X3.135-1986" standard.

# Source Code Transfer Between Machines



## 7.1 INTRODUCTION

One of the major problems inhibiting the porting of applications between UNIX system derivatives is that of incompatible media standards and the physical problems of transferring source code in machine readable form.

The X/OPEN Group takes this problem seriously and has agreed common standards for the transfer of source code. Detailed standards are defined in "SOURCE CODE TRANSFER".

Standards are defined for transfer of 5¼" floppy discs and ½" magnetic tape between machines. Because of the different nature of X/OPEN systems, ranging from single user work stations to large mainframes, it is not possible to define formats which are portable across the whole range. Defining standards for both floppy discs and ½" magnetic tape gives the highest practical coverage of systems.

Current differences in the physical recording formats between cartridge tape devices prevents the definition of a standard for this popular medium.

Because of restrictions imposed by existing hardware, some X/OPEN members are not able to support the floppy disc standard.



## 7.2 FLOPPY DISC STANDARD

As exchange media, the X/OPEN group defines standards for 40 and 80 track floppy discs. It is intended that the prime format should be 80 track, with 40 track retained for compatibility with personal computers. X/OPEN systems equipped only with 80 track disc drives will offer the facility to read 40 track floppy discs by skipping alternate tracks.

## 7.3 MAGNETIC TAPE

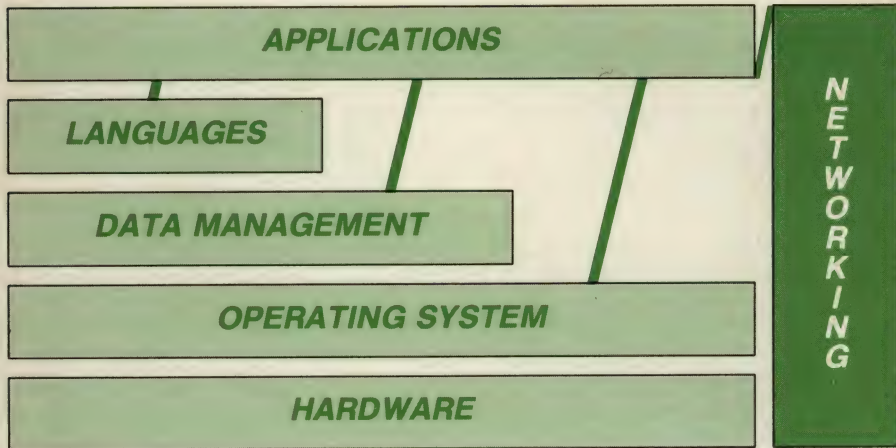
The X/OPEN standard for magnetic tape covers ½" magnetic tape, with a number of different recording formats and densities. The prime format is 9 track Phase Encoded at 1600 bits per inch.

## 7.4 UTILITIES

"SOURCE CODE TRANSFER" includes the definition of two alternative utilities for the archiving of files to the transfer medium and their subsequent retrieval, *tar* and *cpio*.

In addition, guidelines are given on the use of direct machine to machine connection and the *uucp* utility, as a means of transferring files between X/OPEN systems.

# Networking and Communications



## 8.1 NETWORKING AND COMMUNICATION

The general target for computer data communications is interworking between systems of different types from different suppliers. Future X/OPEN definitions for such open systems interworking can be expected to embrace the ISO OSI standard.

Two services specific to systems supporting the System V Interface have been identified. These are "Generalised Inter Process Communication" (IPC) and "Distributed File System".

Many current commercial applications are supported via interactive transaction processing systems. X/OPEN definitions in this area can be expected to be in line with emerging International Standards.



## 8.2 OPEN SYSTEMS INTERCONNECTION

For interworking to be possible, systems must have common methods of describing both tasks and data, and must be capable of functioning in a defined manner. To describe interconnection, an architectural model is required. The International Organisation for Standardisation, ISO, has developed the "Reference Model for Open Systems Interconnection" (IS7498). This is often referred to as the "ISO OSI model" or the "ISO 7-layer model". This model sets a framework into which protocol and service standards can be set.

The ISO OSI target is to have a complete set of protocol and service definitions which comply with the 7-layer model and which are internationally agreed and published as ISO standards.

A complete set of ISO OSI standards does not yet exist. Where standards are available, they contain options. For practical systems to be built, it is necessary to have a very clear definition of standards to be adopted and the options to be used.

To provide a clear statement of the standards to be used, a specialist working group has been formed by the 12 major European vendors who propose the technical objectives for "ESPRIT", the "European Strategic Programme for Research into Information Technology". This is called the "Standards Promotion and Application Group, SPAG".

Where ISO standards do not yet exist, interim standards from one of the national or international standards bodies are adopted by SPAG. Such standards are expected to form the basis of future ISO standards. SPAG is not itself a standard making body. Its recommendations will reflect evolving standards.

X/OPEN member companies are committed to the ISO OSI target and the adoption of ISO standards. The group will monitor SPAG recommendations.

X/OPEN intends to define application interfaces for access to OSI services to ensure the portability of applications and library routines.

### 8.3 GENERALISED INTER-PROCESS COMMUNICATION, IPC

UNIX operating systems provide limited IPC capabilities in the form of "pipes" and "fifos". Kernel extensions within the AT&T System V Interface Definition provide some further IPC mechanisms for the passing of messages between processes in the same memory address space. These extensions were omitted from issue 1 of the X/OPEN definition because it was believed that a much more generalised mechanism for peer to peer communication between processes, either in the same physical machine or in different machines connected via some communications medium is needed.

The X/OPEN group is working on the definition of such a mechanism and but in the short term, it is recognised that there are some applications which need access to such IPC capabilities as currently exist. "XVS INTER-PROCESS COMMUNICATION" gives detailed definitions for

- Message passing between processes.
- Shared memory.
- Semaphores.

It must be recognised that these routines cannot be implemented on all hardware architectures and hence are optional in the X/OPEN definition. However, where the interfaces are supported, the behaviour will be as defined.



**8.4 DISTRIBUTED FILE SYSTEM**

There is an increasing requirement to be able to access data contained within UNIX File Systems on machines connected together by a local area network from any system on that network. The totality of data accessible in this way can be regarded as a "distributed file system".

The X/OPEN Group regards the way in which local resources are made available to other systems to be a matter of system administration, and does not intend to publish detailed definitions.

The only aspect of such systems which is relevant to the application developer is the behaviour of the system towards applications at run-time.

The characteristics of any distributed file system supported by X/OPEN systems are:

- Access to the distributed file system is via the standard input/output system calls, and is identical for local and remote files.
- No changes are necessary to existing applications. Binary copies of existing applications are able to access a distributed file system, subject to the requirement that the data within a file is in a compatible format.
- Naming of remote items follows the same syntax as for local items and no new naming conventions are required.
- File locking applies across the network.

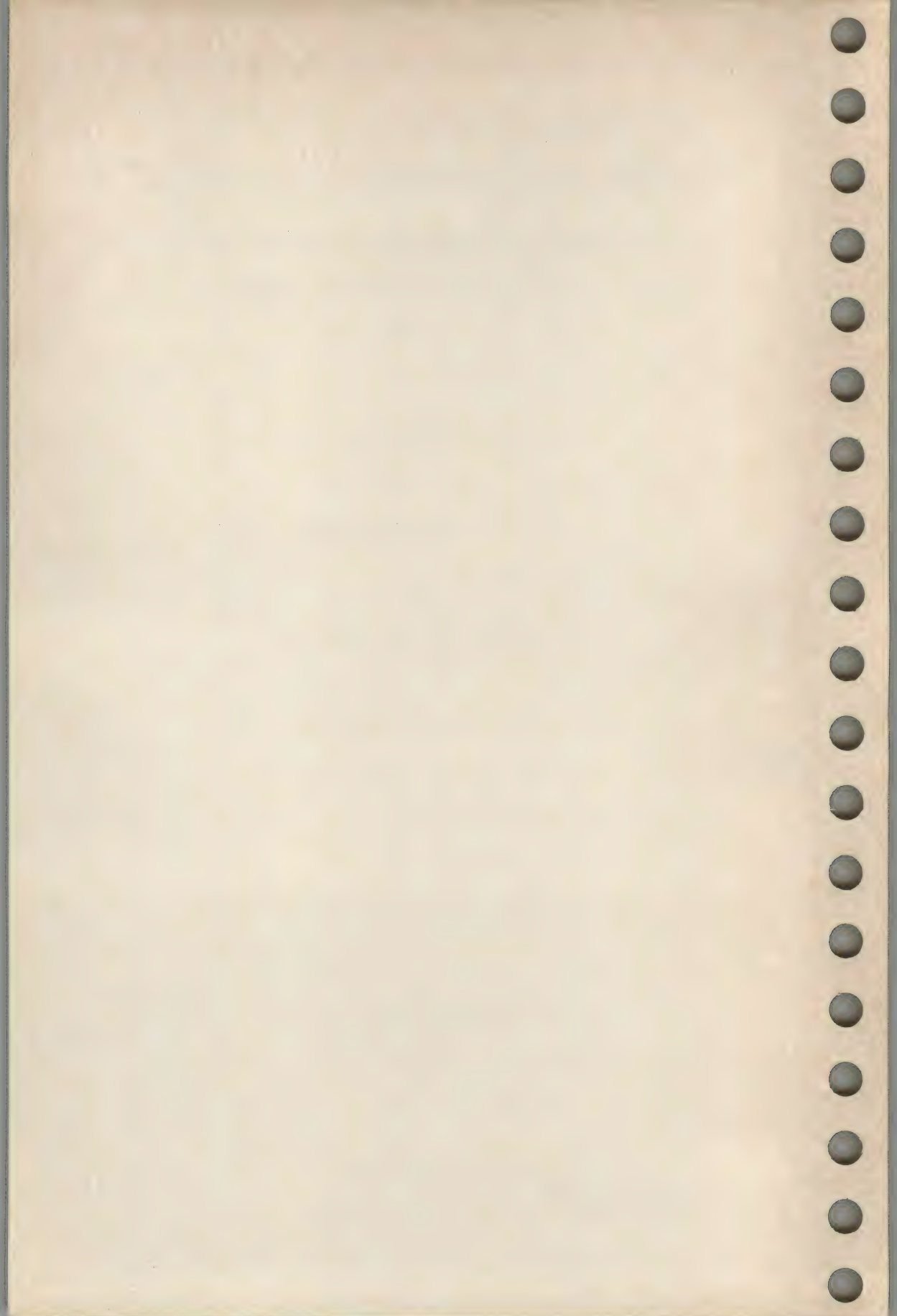
8.5   **DISTRIBUTED TRANSACTION PROCESSING**

Many commercial applications require interactive transaction processing facilities and the X/OPEN Group considers the provision of such facilities to be of key importance.

International Standards Organisations have not made the expected progress in this area.

The X/OPEN Group intends to take action to improve this situation.



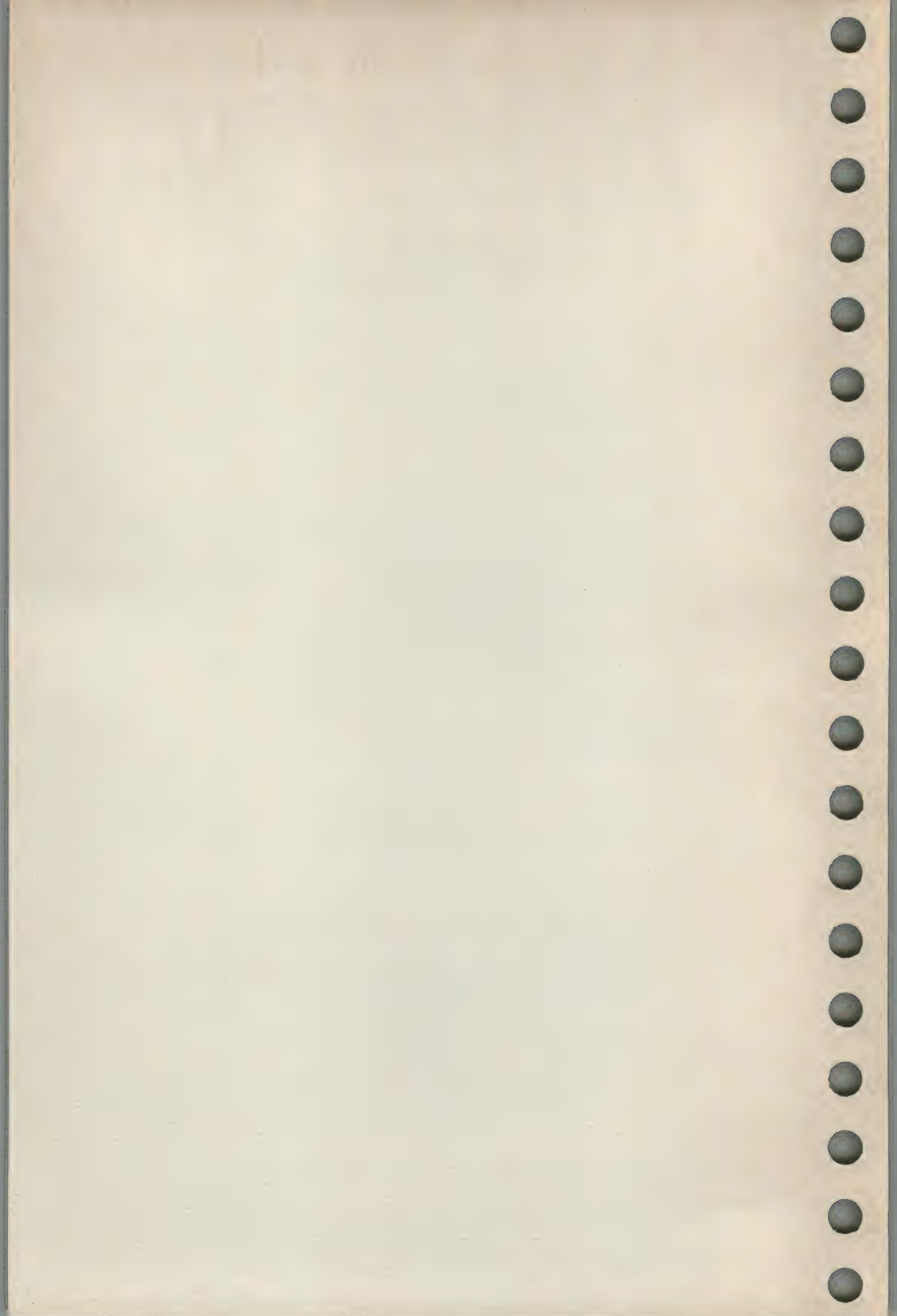


# X/O/P/E/N/

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION  
COMMANDS AND UTILITIES





# **Contents**

Chapter	1	INTRODUCTION
	1.1	OVERVIEW
	1.1.1	Rationale
	1.2	STATUS OF INTERFACES
	1.2.1	Mandatory
	1.2.2	Optional
	1.2.3	Portability
	1.2.4	Relationship to SVID
	1.2.5	Subject to Change
	1.3	FORMAT OF ENTRIES
	1.4	DEFINITIONS
	1.4.1	Character and Block Special Files
	1.4.2	Character Sets
	1.4.3	Command Interpreter
	1.4.4	Directory
	1.4.5	Effective User ID and Effective Group ID
	1.4.6	Fifo Special Files
	1.4.7	File Access Permissions
	1.4.8	File Descriptor
	1.4.9	File Name
	1.4.10	Parent Process ID
	1.4.11	Path Name and Path Prefix
	1.4.12	Process Group ID
	1.4.13	Process Group Leader
	1.4.14	Process ID
	1.4.15	Real User ID and Real Group ID
	1.4.16	Root Directory and Current Working Directory
	1.4.17	Special Processes
	1.4.18	Super-user
	1.4.19	Terminal Group
	1.4.20	Tty Group ID
	1.5	REGULAR EXPRESSIONS
	1.5.1	Simple Regular Expressions
	1.5.2	Extended Regular Expressions
	1.5.3	Precedence
	1.5.4	Examples
	1.6	SHELL METANOTATION
	1.7	SIGNALS



	1.8	DIRECTORY STRUCTURE
	1.9	ENVIRONMENTAL VARIABLES
	1.10	SYSTEM RESIDENT DATA FILES
	1.11	SPECIAL FILES
	1.12	OUTPUT DEVICES
	1.13	SHELL BUILT-INS
	1.14	CAVEATS
	1.14.1	The Files <values.h> and <limits.h>
	1.14.2	Internationalisation
	1.15	EXIT STATUS
	1.16	LIST OF SERVICES
Chapter	2	COMMANDS AND UTILITIES
	2.1	Commands and Utilities
		<i>admin</i> (1D)
		<i>ar</i> (1)
		<i>as</i> (1D)            OPTIONAL
		<i>at</i> (1)
		<i>awk</i> (1)
		<i>banner</i> (1)
		<i>basename</i> (1)
		<i>cal</i> (1)
		<i>calendar</i> (1)
		<i>cat</i> (1)
		<i>cc</i> (1D)
		<i>cd</i> (1)
		<i>cflow</i> (1D)
		<i>chmod</i> (1)
		<i>chown</i> (1)
		<i>chroot</i> (1)
		<i>cmp</i> (1)
		<i>col</i> (1)
		<i>comm</i> (1)
		<i>cp</i> (1)
		<i>cpio</i> (1)
		<i>cpp</i> (1D)
		<i>crontab</i> (1)
		<i>csplit</i> (1)
		<i>cu</i> (1)
		<i>cut</i> (1)
		<i>cxref</i> (1D)
		<i>date</i> (1)
		<i>dd</i> (1)
		<i>delta</i> (1D)

## Contents

<i>df</i> (1)	
<i>diff</i> (1)	
<i>dircmp</i> (1)	
<i>dis</i> (1D)	OPTIONAL
<i>du</i> (1)	
<i>echo</i> (1)	
<i>ed</i> (1)	
<i>egrep</i> (1)	
<i>env</i> (1D)	
<i>ex</i> (1)	
<i>expr</i> (1)	
<i>file</i> (1)	
<i>find</i> (1)	
<i>get</i> (1D)	
<i>grep</i> (1)	
<i>id</i> (1)	
<i>join</i> (1)	
<i>kill</i> (1)	
<i>ld</i> (1D)	
<i>lex</i> (1D)	
<i>line</i> (1)	
<i>lint</i> (1D)	
<i>logname</i> (1)	
<i>lorder</i> (1D)	
<i>lp</i> (1)	
<i>lpstat</i> (1)	
<i>ls</i> (1)	
<i>m4</i> (1D)	
<i>mail</i> (1)	
<i>mailx</i> (1)	OPTIONAL
<i>make</i> (1D)	
<i>mesg</i> (1)	
<i>mkdir</i> (1)	
<i>mknod</i> (1)	OPTIONAL
<i>newgrp</i> (1)	OPTIONAL
<i>news</i> (1)	OPTIONAL
<i>nl</i> (1)	
<i>nm</i> (1D)	
<i>nohup</i> (1)	
<i>od</i> (1)	
<i>pack</i> (1)	
<i>passwd</i> (1)	
<i>paste</i> (1)	
<i>pg</i> (1)	
<i>pr</i> (1)	
<i>prof</i> (1D)	OPTIONAL
<i>prs</i> (1D)	



<i>ps</i> (1)	
<i>pwd</i> (1)	
<i>rm</i> (1)	
<i>rmdel</i> (1D)	
<i>sact</i> (1D)	
<i>sdb</i> (1D)	OPTIONAL
<i>sed</i> (1)	
<i>sh</i> (1)	
<i>shl</i> (1)	OPTIONAL
<i>size</i> (1D)	
<i>sleep</i> (1)	
<i>sort</i> (1)	
<i>spell</i> (1)	
<i>split</i> (1)	
<i>strip</i> (1D)	
<i>stty</i> (1)	
<i>su</i> (1)	
<i>sum</i> (1)	
<i>tabs</i> (1)	
<i>tail</i> (1)	
<i>tar</i> (1)	
<i>tee</i> (1)	
<i>test</i> (1)	
<i>time</i> (1D)	
<i>touch</i> (1)	
<i>tr</i> (1)	
<i>true</i> (1)	
<i>tsort</i> (1D)	
<i>tty</i> (1)	
<i>umask</i> (1)	
<i>uname</i> (1)	
<i>unget</i> (1D)	
<i>uniq</i> (1)	
<i>uucp</i> (1)	
<i>uustat</i> (1)	
<i>uuto</i> (1)	
<i>uux</i> (1)	
<i>val</i> (1D)	
<i>vi</i> (1)	
<i>wait</i> (1)	
<i>wall</i> (1)	
<i>wc</i> (1)	
<i>what</i> (1D)	
<i>who</i> (1)	
<i>write</i> (1)	
<i>xargs</i> (1D)	
<i>yacc</i> (1D)	

# Introduction

## 1.1 OVERVIEW

### 1.1.1 Rationale

The basic X/OPEN System V Specification (XVS) is separated into "XVS COMMANDS AND UTILITIES" and "XVS SYSTEM CALLS AND LIBRARIES".

This part defines Commands and Utilities which are accessed via command interpreters. The System Calls and Library interfaces and the Commands and Utilities are jointly known as services.

The XVS defines the interfaces and run-time behaviour provided by the services to application programs. No particular restrictions are imposed on the way in which the services are implemented.

The Commands and Utilities are defined in terms of their interface as seen from the *sh(1)*<sup>†</sup> command interpreter. Such an interface is also available to application programs through the *exec(2)*, *popen(3S)* and *system(3S)* interfaces.

The Commands and Utilities are all to be found under a single heading. They are split functionally into those which are intended to provide an applications interface (referred to as *standard utilities*) and those which are only intended to be used by development programmers or during the porting of applications to an X/OPEN system (referred to as *development utilities*). The standard utilities will be present in all X/OPEN systems, as they are needed to provide a run-time interface to applications. The development utilities may only be present in development systems; their descriptions are carefully annotated to indicate this.

---

<sup>†</sup> This notation is described in the "Format of Entries" section.



## 1.2 STATUS OF INTERFACES

### 1.2.1 Mandatory

The majority of the Commands and Utilities are mandatory; they must be present in all X/OPEN systems and they must conform to the published definition.

The development utilities may not be present in all X/OPEN systems; in designated "development" systems all of the development utilities must be present and must conform to the published definition with the exception of those identified below.

### 1.2.2 Optional

A small number of utilities are optional. The presence of these is not mandatory, although if they are present they must conform to the definition. The list below shows those which are optional.

Optional Utilities	
<i>as</i> (1D)	<i>news</i> (1)
<i>dis</i> (1D)	<i>prof</i> (1D)
<i>mailx</i> (1)	<i>sdb</i> (1D)
<i>mknod</i> (1)	<i>shl</i> (1)
<i>newgrp</i> (1)	

In addition, there are a number of utilities which may not be supportable on X/OPEN systems (see **Portability**, below).

### 1.2.3 Portability

Commands and Utilities offer functionality and interfaces that are substantially more complex than that of the System Calls and Library Routines. They are in many cases much harder to define and it is therefore more difficult to guarantee portability of interfaces and consistent behaviour between systems.

The definition of standards for commands and utilities is an evolving process and X/OPEN intends to participate fully.

The current definition is a valuable first step, but additional work will be needed to evolve towards a complete standard. For example:

- The number of options defined for many of the commands is excessive and includes functionality which is rarely used or is implementation specific.
- There are too many different ways of achieving the same results.
- Many of the current descriptions were written to record the observed behaviour of already existing utilities and the level of precision is inadequate for use as a definitive standard.

Rectification of this is an enormous task and the current X/OPEN definition is of necessity based on the available documentation, but incorporates annotation to

highlight potential portability problems resulting from the points listed above.

X/OPEN is working on a substantially improved definition of commands, with the number of options reduced to those in common use and with a higher level of specification. "XVS COMMANDS AND UTILITIES APPENDIX A" contains a proposed template for improved specifications together with a number of examples.

This improvement process will be carried out in close consultation with the various user organisations and standards bodies, such as IEEE and ISO, to ensure that the result is a single standard definition of operating system commands and utilities.

Readers of the X/OPEN Portability Guide are invited to participate directly in the consultative process to ensure that the evolving standard matches the requirements of existing and future applications. The preface to this volume of the Portability Guide includes details of how to submit comments directly to the X/OPEN Group.

Wherever reduced functionality is highlighted to warn that reduced portability exists, the reason is shown by a code in the right-hand margin which refers to the table given below. In all cases an application wishing to achieve maximum portability should avoid such functionality.

Unless the primary task of a utility is to produce textual material on its standard output, application developers should not rely on the format or content of any such material which may be produced. Where the primary task *is* to provide such material, but the output format is incompletely specified, the description is marked. Applications developers are warned not to expect that the output of such a utility on one system will be any guide to its behaviour on another system.

It is expected that the output from Software Development Utilities will be read by an expert user who can interpret it, and no warning is given where their output format is incompletely specified or has a language or cultural dependency.

The warning codes and their meanings are as follows:

OF Output format incompletely specified.

The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

LA Language or cultural dependency.

The output produced by the utility is in a language or format which may not be understood by all users.

OP Dependent on optional service in XVS.

Typical implementations depend on an optional service and the functionality affected may not be present if the optional service is not supported.

UN Possibly unsupported feature.

It may not be possible to implement the required functionality (as defined) on all X/OPEN systems and the functionality may not be present. This may, for example, be the case where the X/OPEN system is hosted by a supervisory régime, which provides the service in an alternative way.



PI The behaviour cannot be guaranteed to be consistent.

It is not possible to guarantee that the utility behaves in the same way on all X/OPEN systems. This is the case if it provides functionality which is system-defined or system-specific. Options which are used to *select* alternative forms of system-specific behaviour are not marked, as it is clear from their descriptions that their use is inherently non-portable.

LV Level 2.

The utility is marked as "Level 2" in the SVID. This means that the utility may no longer be supported after three years. The date at which the three year period started is shown.

MV Marginal value.

The functionality described is of marginal value and may be removed in future editions of the Guide.

#### 1.2.4 Relationship to SVID

Volume 2 of the System V Interface Definition (SVID), published by AT&T, in Spring 1986 includes definitions of Commands and Utilities. The definition groups interfaces into a mandatory base plus a series of extensions.

The X/OPEN Commands and Utilities Definition corresponds to all of the SVID Base Utilities, Advanced Utilities and Software Development Utilities. The X/OPEN definition does not differentiate between the base and advanced utilities; they are treated as one group, the Standard Utilities.

Except for a number of cases where the opportunity has been taken to clarify some explanations, wherever an X/OPEN definition differs from the corresponding one in the SVID, the differences are noted in the description.

#### 1.2.5 Subject to Change

The SVID identifies certain services as possibly subject to withdrawal, by referring to them as "Level 2" services. These are shown in the following table.

Services subject to change	
	Service Valid until
<i>egrep</i> (1)	November 30 1988
<i>fgrep</i> (1)	"

The X/OPEN commitment to support of these services matches that of the SVID.

**1.3      FORMAT OF ENTRIES**

The entries in each chapter are based on a common format, not all of whose parts always appear.

Each entry describes one or more of the Commands and Utilities. Each page is identified at the top outer corner by the entry name. In accordance with common practice, each entry is given a suffix to indicate its category. Standard Utilities are suffixed (1); Development Utilities are suffixed (1D).

The formal description consists only of the **NAME**, **SYNOPSIS**, **DESCRIPTION**, **FILES**, **ERRORS** and **EXIT STATUS** parts.

**NAME** gives the name or names of the services described by the entry and briefly states their purpose.

**SYNOPSIS** summarises the use of the service being described.

**DESCRIPTION** defines the functionality and behaviour of the service.

**EXAMPLE(S)** gives example(s) of usage, where appropriate.

**FILES** gives the names of files used.

**ERRORS** describes conditions which may give rise to an error.

**EXIT STATUS** is shown if a useful exit status is provided.

**SEE ALSO** lists related entries.

**APPLICATION USAGE** gives additional information about the way that the utility should be used and any other relevant information.

**FUTURE DIRECTIONS** should be used as a guide to current thinking; there is not necessarily a commitment to implement all of these future directions in their entirety.

**CHANGE HISTORY** shows the derivation and a list of differences between issues, with reasons for each change.



The following typographical conventions are used:

**Boldface** strings are literals and are to be typed just as they appear.

*Italic* strings usually represent substitutable argument prototypes and the names of entries found elsewhere.

Names in upper case surrounded by braces, e.g. {CONST} represent system-dependent constants.

Square brackets [] around an argument prototype indicate that the argument is optional. When an argument prototype is given as *name* or *file*, it always refers to a path name.

Ellipses ... are used to show that the previous argument may be repeated.

Whenever referring to a service described elsewhere, the name includes the suffix used to classify entries, in accordance with standard practice. Suffices 2 - 7 indicate interfaces described in "XVS SYSTEM CALLS AND LIBRARIES". Suffix 1 refers to a Command or Utility defined in this part of the Guide.

## 1.4 DEFINITIONS

Many special terms are used in the interface definitions. The descriptions of these terms follows.

### 1.4.1 Character and Block Special Files

Character and block special files are used to refer to physical devices. Certain restrictions may apply to use of character and block special files which are implementation dependent.

### 1.4.2 Character Sets

Many of the services refer to the ASCII or ISO 646 character set.

For a full definition refer to "XVS INTERNATIONALISATION".

### 1.4.3 Command Interpreter

It is possible for applications to invoke utilities through a number of interfaces, which are collectively considered to act as command interpreters. The most obvious of these are *sh*(1) and *system*(3S), although *popen*(3S) and the various forms of *exec*(2) may also be considered to behave as interpreters.

### 1.4.4 Directory

Directories organise files into a hierarchical system where directories are the nodes of the hierarchy. A directory is a file that catalogues the list of files, including directories (sub-directories), that are directly beneath it in the hierarchy. Entries in a directory file are called links, which associate a file identifier with a file name. By convention, a directory contains at least two links, *.* and *..*, referred to as *dot* and *dot-dot* respectively. *Dot* refers to the directory itself and *dot-dot* refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The path-name of the root directory is */* and the parent directory of the root directory is */*.

### 1.4.5 Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group-ID bit set, see *exec*(2). In addition, they can be changed with *setuid*(2) and *setgid*(2), respectively.

### 1.4.6 Fifo Special Files

A fifo special file is a named "pipe", see *mknod*(1), *pipe*(2) and *mknod*(2). Normally, a fifo special file is opened in conjunction by two or more separate processes. One or more processes write data to the fifo special file and another process reads this same data from the file on a "first-in-first-out" basis. Seeks on a fifo special file have no meaning and cause the [ESPIPE] error.



#### 1.4.7 File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user, and in the case of execute permission, at least one of the execute bits is set.

The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (S\_IRWXU) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (S\_IRWXG) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (S\_IRWXO) of the file mode is set.

Otherwise, the corresponding permissions are denied.

#### 1.4.8 File Descriptor

A file descriptor is a small integer used to identify a file for the purpose of doing I/O. The value of a file descriptor is from 0 to {OPEN\_MAX}-1. A process may have no more than {OPEN\_MAX} file descriptors open simultaneously.

A file descriptor has associated with it information used in performing I/O on the file: a file pointer that marks the next position within the file where I/O will begin; file status and access modes (e.g. read, write, read/write), see *open(2)*; and close-on-exec flag, see *fcntl(2)*. Multiple file descriptors may identify the same file. A file descriptor is returned by system routines such as *creat(2)*, *dup(2)*, *fcntl(2)*, *open(2)*, or *pipe(2)*. The file descriptor is used as an argument by routines such as *read(2)*, *write(2)*, *ioctl(2)* and *close(2)*.

#### 1.4.9 File Name

Names consisting of 1 to {NAME\_MAX} characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding the characters "null" and "slash".

Note that it is generally unwise to use \*, ?, !, [, or ] as part of file names because of the special meaning attached to these characters for filename expansion by some command interpreters, see *system(3S)*. Other characters to avoid are the hyphen, blank, tab, <, >, backslash, single and double quotes, accent grave, vertical bar, carat, curly braces and parentheses. It is also advisable to avoid the use of non-printing characters in file names.

#### 1.4.10 Parent Process ID

A new process is created by a currently active process, see *fork(2)*. The parent process ID of a process is the process ID of its creator, unless the creator has already exited, see *exit(2)*.

#### 1.4.11 Path Name and Path Prefix

In a C program a path name is a null-terminated character-string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name. The null string is undefined and may be considered an error.

More precisely, a path name is a null-terminated character string constructed as follows:

```
<path-name>::= <file-name>| <path-prefix><file-name>| /|.|..  
<path-prefix>::= <rtprefix>| /<rtprefix>| /  
<rtprefix>::= <dirname>| <rtprefix><dirname> /
```

where <file-name> is a string of 1 to {NAME\_MAX} characters other than slash and null, and <dirname> is a string of 1 to {NAME\_MAX} characters (other than slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory. The meanings of . and .. are defined under *Directory*.

The result of names not produced by the grammar is undefined.

#### 1.4.12 Process Group ID

Each active process is a member of a process group. The process group is uniquely identified by a non-negative integer, called the process group ID, which is the process ID of the group leader (see below). This grouping permits the signaling of related processes, see *kill(2)*. This grouping is also used to terminate a group of related processes upon termination of the group leader, see *exit(2)* and *signal(2)*.

#### 1.4.13 Process Group Leader

A process group leader is any process whose process group ID is the same as its process ID. Any process may detach itself from its current process group and become a process group leader by calling *setpgid(2)*. A process inherits the process group ID of the process that created it, see *fork(2)* and *exec(2)*.

#### 1.4.14 Process ID

Each active process in the system is uniquely identified by a non-negative integer called a process ID. The range of this ID is from 0 to {PID\_MAX}. Process IDs between 0 and {SYSPID\_MAX} inclusive are reserved for special system processes.



#### 1.4.15 Real User ID and Real Group ID

Each user allowed on the system is identified by a non-negative integer called a real user ID.

Each user is also a member of a group. The group is identified by a non-negative integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process. They can be reset with *setuid(2)* and *setgid(2)*, respectively.

#### 1.4.16 Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system, see *chroot(2)*.

#### 1.4.17 Special Processes

Special processes are system processes as, for example, a system's process scheduler. Process IDs between 0 and {SYSPID\_MAX} inclusive are reserved for special system processes.

#### 1.4.18 Super-user

A process is recognised as a *super-user* process and is granted special privileges if its effective user ID is 0.

#### 1.4.19 Terminal Group

A terminal group is a group of processes associated with a particular terminal. The group is used to select those processes which will receive signals generated by the terminal handling part of a system. In particular, SIGHUP, SIGINT and SIGQUIT are commonly caused in this way. On asynchronous lines (see *termio(7)*) the latter two are generated on receipt of the INTR and QUIT characters, if the appropriate mode has been enabled. This grouping is also used to terminate a group of related processes upon termination of the group leader, see *exit(2)* and *signal(2)*.

#### 1.4.20 Tty Group ID

The tty group ID is a non-negative integer used to identify a terminal group.

## 1.5 REGULAR EXPRESSIONS

Regular Expressions are used widely throughout the services. The regular expression is a powerful mechanism for locating and manipulating patterns in text.

The following Commands and Utilities make use of one of the forms of regular expressions:

Utilities using Regular Expressions	
<i>awk</i> (1)	<i>csplit</i> (1)
<i>ed</i> (1)	<i>egrep</i> (1)
<i>ex</i> (1)	<i>expr</i> (1)
<i>grep</i> (1)	<i>lex</i> (1D)
<i>pg</i> (1)	<i>sdb</i> (1D)
<i>sed</i> (1)	<i>vi</i> (1)

A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be **matched** by the RE.

A *pattern* is constructed from one or more REs. An RE consists of either *ordinary characters* or *metacharacters*.

Within a pattern, all alphanumeric characters match themselves; that is to say, the RE pattern "*abc*", when applied to a set of strings, will only match those strings containing the character sequence "*abc*" anywhere in them.

Most other characters also match themselves; however, a small set are known as the metacharacters. These characters have special meanings when encountered in patterns. They are described below.

There are basically two forms of regular expression: *simple* and *extended*. The sort of RE used by a utility is indicated in the description of that utility.

### 1.5.1 Simple Regular Expressions

The simple regular expressions are constructed as follows:

Expression	Meaning
------------	---------

<i>c</i>	The character <i>c</i> where <i>c</i> is not a special character.
<i>\c</i>	The character <i>c</i> where <i>c</i> is any character with special meaning, see below.
<i>^</i>	The beginning of the string being compared.
<i>\$</i>	The end of the string being compared.
<i>.</i>	Any character.
<i>[s]</i>	Any character in the non-empty set <i>s</i> , where <i>s</i> is a sequence of characters. Ranges may be specified as <i>c-c</i> . The character <i>]</i> may be included in the set by placing it first in the set. The character <i>-</i> may be included in the set by placing it first or last in the set. The



	character <code>^</code> may be included in the set by placing it anywhere other than first in the set, see below.
<code>[^s]</code>	Any character not in the set <code>s</code> , where <code>s</code> is defined as above.
<code>r*</code>	Zero or more successive occurrences of the regular expression <code>r</code> . The longest leftmost match is chosen.
<code>rx</code>	The occurrence of regular expression <code>r</code> followed by the occurrence of regular expression <code>x</code> . (Concatenation)
<code>r\{m,n\}</code>	Any number of <code>m</code> through <code>n</code> successive occurrences of the regular expression <code>r</code> . The regular expression <code>r\{m\}</code> matches exactly <code>m</code> occurrences <code>r\{m,\}</code> matches at least <code>m</code> occurrences. The maximum number of occurrences is matched.
<code>\(r\)</code>	The regular expression <code>r</code> . The <code>\(</code> and <code>\)</code> sequences are ignored.
<code>\n</code>	When <code>\n</code> (where <code>n</code> is a number in the range 1 to 9) appears in a concatenated regular expression, it stands for the regular expression <code>x</code> where <code>x</code> is the <code>n</code> th regular expression enclosed in <code>\(</code> and <code>\)</code> sequences that appeared earlier in the concatenated regular expression. For example, in the pattern <code>\(r\)\x\(\y\)\z\2</code> , the <code>\2</code> matches the regular expression <code>y</code> , giving <code>rxzyz</code> .

Characters that have special meaning except where they appear within square brackets, `[ ]`, or are preceded by `\` are: `.`, `*`, `[`, `\`. Other special characters, such as `$` have special meaning in more restricted contexts.

The character `^` at the beginning of an expression permits a successful match only immediately after a new-line or at the beginning of each of the strings to which the match is applied, and the character `$` at the end of an expression requires a trailing new-line.

Two characters have special meaning only when used within square brackets. The character `-` denotes a range, `[c-c]`, unless it is just after the open bracket or before the closing bracket, `[c-]` or `[c]` in which case it has no special meaning. When used within brackets, the character `^` has the meaning *complement of* if it immediately follows the open bracket, `[^c]`, elsewhere between brackets, `[c^]`, it stands for the ordinary character `^`.

The special meaning of the `\` operator can be escaped *only* by preceding it with another `\`, e.g., `\\`.

### 1.5.2 Extended Regular Expressions

Extended regular expressions make use of the same metacharacters as the simple regular expressions, with the following modifications:

1. The `+`, when following an RE, is similar to the `*`. In this case, the RE is matched one or more times.
2. The `?`, when following an RE, causes the RE to be matched zero or one times.

3. The notation  $\backslash(\dots\backslash)$  of simple regular expressions has no special meaning.
4. An RE enclosed between parentheses, ( and ) is an RE that matches whatever the unadorned RE matches.
5. REs separated by the vertical bar, |, are alternatives. As an example, the pattern  $((RE1RE2)|RE3)RE4$  will match any strings containing either  $RE1RE2RE4$  or  $RE3RE4$ . This is known as alternation.

### 1.5.3 Precedence

The precedence of the operators is as shown below:

Operator Precedence	
[...]	high precedence
* ? +	
concatenation	
alternation	low precedence

### 1.5.4 Examples

Below are examples of regular expressions:

SIMPLE R.E.s	
Pattern	Meaning
ab.d	a b any character d
ab.*d	a b any sequence of characters (including none) d
ab[xyz]d	a b either x, y or z d
ab[~c]d	a b anything except c d
^abcd\$	a line containing only a b c d

EXTENDED R.E.s	
Pattern	Meaning
ab.+d	a b any sequence of one or more characters d
abc?d	either a b c d or a b d
(abc xyz)	either a b c or x y z



## 1.6 SHELL METANOTATION

The shell metanotation characters are similar to regular expressions. The important difference is that the shell metanotation characters are used to match *file names*.

The following metacharacters are defined:

- ★ Matches any string, including the null string.
- ? Matches any single character.
- [...] Matches any one of the enclosed characters. A pair of characters separated by - matches any character lexically between the pair, inclusive. If the first character following the opening [ is a !, any character not enclosed is matched.

These characters are used by the following Commands and Utilities:

Utilities using Shell Metanotation	
<i>cpio</i> (1)	<i>find</i> (1)
<i>sh</i> (1)	<i>uux</i> (1)

## 1.7 SIGNALS

To be portable, applications should only send, catch or ignore the following signals:

Signal	Description
SIGHUP	hangup
SIGINT	interrupt (rubout)
SIGQUIT	quit
SIGILL	illegal instruction (not reset when caught)
SIGTRAP	trace trap (not reset when caught)
SIGABRT†	process abort signal
SIGFPE	floating point exception
SIGKILL	kill (cannot be caught or ignored)
SIGSYS	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal from kill
SIGUSR1	user defined signal 1
SIGUSR2	user defined signal 2

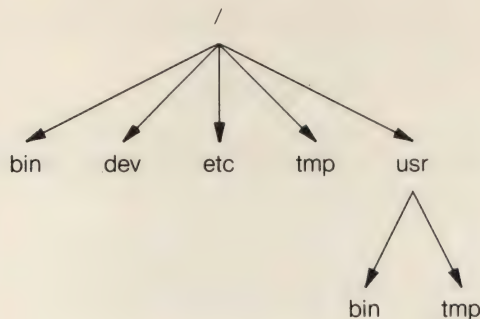
The signal marked above **SIGABRT†** has been included from a FUTURE DIRECTION indicated in the SVID.

It should be noted that in some cases signals can be sent to processes as a result of keystrokes on terminals. See the definition of *Terminal Group*.



## 1.8 DIRECTORY STRUCTURE

Below is a diagram of the minimal directory tree structure present on any system.



The following guidelines apply to the contents of these directories:

**/bin, /dev, /etc** and **/tmp** are primarily for the use of the system. Most applications should never create files in any of these directories, though they may read and execute them.

**/bin** contains executable system commands (utilities), although it may be empty.

**/dev** contains special files (I/O devices). Those which are always present in X/OPEN systems are defined in "XVS SYSTEM CALLS AND LIBRARIES".

**/etc** contains system data files such as **/etc/passwd**. It may also contain some executable files which are used by the system; these are not intended to be accessible to the ordinary user.

**/tmp** contains temporary files created by any utilities in **/bin**, and other system processes. Applications should use **/usr/tmp**.

**/usr/bin** and **/usr/tmp** can be used by applications as well as the system.

**/usr/bin** contains (user-level) executable application commands and system commands.

**/usr/tmp** contains temporary files created by application programs and by the system.

If the system is re-started after being halted for any reason, applications cannot rely on the contents of **/tmp** or **/usr/tmp** remaining undisturbed. It is common for both directories to be emptied by the restart procedures.

## 1.9 ENVIRONMENTAL VARIABLES

An array of strings is made available by *exec*(2) when a process begins execution, see also *system*(3S). These strings are described more fully in *environ*(5), but the minimum set of strings that can be expected to exist and be set in any X/OPEN environment are:

Variable	Use
HOME	Full pathname of the user's home directory, set when the user signs on.
LOGNAME	Login name.
PATH	A colon separated ordered list of pathnames that determine the search sequence used in locating files.
TERM	The kind of terminal for which output is prepared.
TZ	Time zone information. See also <i>ctime</i> (3C).

The internationalisation extension defines additional environmental variables. Full details are given in "XVS INTERNATIONALISATION".



## 1.10 SYSTEM RESIDENT DATA FILES

The only system-resident data files implied by the X/OPEN system definition are given in the following table, together with a reference to the appropriate chapter and entry to find their definitions.

data file	reference
/etc/group	<i>group</i> (4)
/etc/passwd	<i>passwd</i> (4)
/etc/utmp	<i>utmp</i> (4)
/etc/profile	<i>see below</i>
/etc/wtmp	<i>utmp</i> (4)

*/etc/profile* is a file of *sh*(1) commands which is used to establish an environment for users when they initially sign-on to an X/OPEN system. The mechanism for the signing-on procedure is not defined, nor is it necessarily the case that users will use *sh*(1) as their command interpreter. However, the file will contain an assignment to the **PATH** environment variable. This is a fundamental requirement to enable the command interpreter in use to locate the utilities in the file system.

## 1.11 SPECIAL FILES

The names of specific I/O devices are known as "special files". The only ones present in every X/OPEN system are given below, together with reference to their descriptions. The ones marked (†) are only present in systems supporting the source code transfer standard and need *not* be present in every system.

device	reference
/dev/console	<i>console</i> (7)
/dev/null	<i>null</i> (7)
/dev/tty	<i>tty</i> (7)
/dev/sctfdm	<i>sct</i> (7)†
/dev/sctmtm	<i>sct</i> (7)†
/dev/rsctfdm	<i>sct</i> (7)†
/dev/rsctmtm	<i>sct</i> (7)†



## 1.12 OUTPUT DEVICES

Many of the Commands and Utilities are intended to be used directly by a user, rather than having their output processed by other programs.

Such utilities may make assumptions regarding the abilities of the terminal in use. For example, *pr(1)* expects the terminal to have a bell.

In some cases it is assumed that an asynchronous terminal is in use. On some systems, block-mode, synchronous terminals will cause those utilities anticipating asynchronous terminals to function in a modified manner, or possibly not at all. Because of the wide variety of terminals in use and the fact that the X/OPEN definition does not constrain the way in which any command is implemented, it is not possible to identify the commands affected in this way.

Application writers should not expect all terminals to support all possible attributes. An application wishing to use a wide set of attributes may restrict the number of terminals it can make use of.

## 1.13 SHELL BUILT-INS

Some commands are built into the shell, *sh*(1). Such commands have been marked in an **APPLICATION USAGE** on the relevant page.

These built-ins may not exist as separate executable commands accessible through the *system*(3S), *popen*(3S) or *exec*(2) interfaces. Some, for example *cd*(1), cannot be implemented as anything other than built-ins.

The following are the relevant commands:

Shell built-in commands	
<i>cd</i> (1)	<i>echo</i> (1)
<i>newgrp</i> (1)†	<i>pwd</i> (1)
<i>test</i> (1)	<i>umask</i> (1)
<i>wait</i> (1)	

† It should also be noted that, although not actually a built-in, on some implementations the command *newgrp*(1) behaves like a built-in. Reference should be made to the entry for this command for further details.



## 1.14 CAVEATS

### 1.14.1 The Files `<values.h>` and `<limits.h>`

A number of limits and values which are of importance to system developers are system dependent. Their values are available in the include files `<limits.h>` and `<values.h>`, as symbolic constants. These constants are referred to in many places in the interface definitions; for example `{SYSPID_MAX}`. The file `<values.h>` is part of the SVID base. The file `<limits.h>`, part of the IEEE trial use standard, defines additional values and has also been included in the XVS.

In some cases, the same value is defined in both of these files, although the name used to refer to the value differs. In X/OPEN systems, both names are set to the same value.

Applications developers may choose to include either, neither or both of these files in a particular application, depending upon their needs. The interfaces defined in the following chapters can always be used without it being necessary to include either of these files.

### 1.14.2 Internationalisation

When using the facilities defined in "XVS INTERNATIONALISATION" it is important that any utilities invoked will process data in a manner which is "8-bit clean". Not all utilities necessarily satisfy this requirement. A list of utilities which are 8-bit clean is given in "XVS INTERNATIONALISATION". In addition, they are marked in the table at the end of this chapter.

## 1.15 EXIT STATUS

Although it is possible for the utilities to return exit values, in general these are not specified and no useful information can be gained from them, unless the process was terminated abnormally by a signal, see *wait(2)*.

Where the utilities do return values with specific meanings, these are documented in the appropriate descriptions. In such cases zero is used to indicate "success" and non-zero to indicate some form of failure.



## 1.16 LIST OF SERVICES

A list of all services specified in the XVS follows. Some of the entries describe a number of services, so to find a particular one knowing its name, look in the column labeled "Service". The entry describing the service is given in the column on its right. For example, both *isascii* and *isspace* are to be found under the overall heading of *ctype(3C)*.

Services marked with a (†) are optional. Services marked with a (§) are affected by internationalisation. Utilities marked with a (□) are 8-bit clean in systems which conform to "XVS INTERNATIONALISATION". System calls marked with a (●) are defined in "XVS INTER-PROCESS COMMUNICATION".

The column marked SVID shows where the corresponding SVID entry is located.

Interface	Entry	SVID
abort	abort (3C)	BA_OS
abs	abs (3C)	BA_LIB
access	access (2)	BA_OS
acct	acct (2)†	KE_OS
acct	acct (4)	
acct	acct (5)	
acos	trig (3M)†	BA_LIB
admin	admin (1D)	SD_CMD
alarm	alarm (2)	BA_OS
ar	ar (1)	BU_CMD
as	as (1D)†	SD_CMD
asctime	ctime (3C)‡	BA_LIB
asin	trig (3M)†	BA_LIB
assert	assert (3X)	SD_LIB
assert	assert (5)	
at	at (1)	AU_CMD
atan	trig (3M)†	BA_LIB
atan2	trig (3M)†	BA_LIB
atof	strtod (3C)‡	BA_LIB
atoi	strtol (3C)	BA_LIB
atol	strtol (3C)	BA_LIB
awk	awk (1)	BU_CMD
banner	banner (1)	BU_CMD
basename	basename (1)	BU_CMD
batch	at (1)	AU_CMD
brk	brk (2)†	
bsearch	bsearch (3C)	BA_LIB
cal	cal (1)	BU_CMD
calendar	calendar (1)	BU_CMD
calloc	malloc (3X)	BA_OS
cancel	lp (1)	AU_CMD
cat	cat (1)□	BU_CMD
cc	cc (1D)	SD_CMD
cd	cd (1)□	BU_CMD
ceil	floor (3M)†	BA_LIB
cflow	cflow (1D)	SD_CMD
chdir	chdir (2)	BA_OS
chgrp	chown (1)	AU_CMD
chmod	chmod (1)	BU_CMD
chmod	chmod (2)	BA_OS
chown	chown (1)	AU_CMD
chown	chown (2)	BA_OS

Interface	Entry	SVID
chroot	chroot (1)	SD_CMD
chroot	chroot (2)†	KE_OS
clearerr	ferror (3S)	BA_OS
clock	clock (3C)	BA_LIB
close	close (2)	BA_OS
closedir	directory (3X)	
cmp	cmp (1)	BU_CMD
col	col (1)	BU_CMD
comm	comm (1)	BU_CMD
console	console (7)	BA_ENV
cos	trig (3M)†	BA_LIB
cosh	sinh (3M)†	BA_LIB
cp	cp (1)□	BU_CMD
cpio	cpio (1)□	BU_CMD
cpio	cpio (4)	
cpp	cpp (1D)	SD_CMD
creat	creat (2)	BA_OS
crontab	crontab (1)	AU_CMD
crypt	crypt (3C)	BA_LIB
csplit	csplit (1)	AU_CMD
ctermid	ctermid (3S)	BA_LIB
ctime	ctime (3C)‡	BA_LIB
ctype	ctype (5)	
cu	cu (1)	AU_CMD
cuserid	cuserid (3S)	
cut	cut (1)	BU_CMD
cxref	cxref (1D)	SD_CMD
date	date (1)	BU_CMD
daylight	ctime (3C)‡	BA_LIB
dd	dd (1)	AU_CMD
delta	delta (1D)	SD_CMD
df	df (1)	BU_CMD
diff	diff (1)	BU_CMD
dircmp	dircmp (1)	AU_CMD
dirent	dirent (5)	
dirname	basename (1)	BU_CMD
dis	dis (1D)†	SD_CMD
drand48	drand48 (3C)	BA_LIB
du	du (1)	BU_CMD
dup	dup (2)	BA_OS
echo	echo (1)□	BU_CMD
ecvt	ecvt (3C)‡	



Interface	Entry	SVID
ed	ed(1)□	BU_CMD
edata	end(3C)†	
egrep	egrep(1)	AU_CMD
encrypt	crypt(3C)	BA_LIB
end	end(3C)†	
endgrent	getgrent(3C)	SD_LIB
endpwent	getpwent(3C)	SD_LIB
endutent	getut(3C)	SD_LIB
env	env(1D)	SD_CMD
erand48	drand48(3C)	BA_LIB
erf	erf(3M)†	BA_LIB
erfc	erf(3M)†	BA_LIB
errno	errno(5)	BA_ENV
errno	perror(3C)	BA_LIB
etext	end(3C)†	
ex	ex(1)	AU_CMD
execl	exec(2)	BA_OS
execle	exec(2)	BA_OS
execvp	exec(2)	BA_OS
execv	exec(2)	BA_OS
execve	exec(2)	BA_OS
execvp	exec(2)	BA_OS
_exit	exit(2)	BA_OS
exit	exit(2)	BA_OS
exp	exp(3M)†	BA_LIB
expr	expr(1)□	BU_CMD
fabs	floor(3M)†	BA_LIB
false	true(1)	BU_CMD
fclose	fclose(3S)	BA_OS
fcntl	fcntl(2)	BA_OS
fcntl	fcntl(5)	
fcvt	ecvt(3C)‡	
fdopen	fopen(3S)	BA_OS
feof	ferror(3S)	BA_OS
ferror	ferror(3S)	BA_OS
fflush	fclose(3S)	BA_OS
fgetc	getc(3S)	BA_LIB
fgetgrent	getgrent(3C)	SD_LIB
fgetpwent	getpwent(3C)	SD_LIB
fgets	gets(3S)	BA_LIB
fgrep	egrep(1)	AU_CMD
file	file(1)	BU_CMD

Interface	Entry	SVID
fileno	ferror(3S)	BA_OS
find	find(1)□	BU_CMD
floor	floor(3M)†	BA_LIB
fmod	floor(3M)†	BA_LIB
fopen	fopen(3S)	BA_OS
fork	fork(2)	BA_OS
fprintf	printf(3S)‡	BA_LIB
fputc	putc(3S)	BA_LIB
fputs	puts(3S)	BA_LIB
fread	fread(3S)	BA_OS
free	malloc(3X)	BA_OS
freopen	fopen(3S)	BA_OS
frexp	frexp(3C)	BA_LIB
fscanf	scanf(3S)‡	BA_LIB
fseek	fseek(3S)	BA_OS
fstat	stat(2)	BA_OS
ftell	fseek(3S)	BA_OS
ftw	ftw(3C)	BA_LIB
ftw	ftw(5)	
fwrite	fread(3S)	BA_OS
gamma	gamma(3M)†	BA_LIB
gcvt	ecvt(3C)‡	
get	get(1D)	SD_CMD
getc	getc(3S)	BA_LIB
getchar	getc(3S)	BA_LIB
getcwd	getcwd(3C)	BA_OS
getegid	getuid(2)	BA_OS
getenv	getenv(3C)	BA_LIB
geteuid	getuid(2)	BA_OS
getgid	getuid(2)	BA_OS
getgrent	getgrent(3C)	SD_LIB
getgrgid	getgrent(3C)	SD_LIB
getgrnam	getgrent(3C)	SD_LIB
getlogin	getlogin(3C)	SD_LIB
getopt	getopt(3C)	BA_LIB
getpass	getpass(3C)	SD_LIB
getpgrp	getpid(2)	BA_OS
getpid	getpid(2)	BA_OS
getppid	getpid(2)	BA_OS
getpw	getpw(3C)	
getpwent	getpwent(3C)	SD_LIB
getpwnam	getpwent(3C)	SD_LIB

Interface	Entry	SVID
getpwuid	getpwent(3C)	SD_LIB
gets	gets(3S)	BA_LIB
getuid	getuid(2)	BA_OS
getutent	getut(3C)	SD_LIB
getutid	getut(3C)	SD_LIB
getutline	getut(3C)	SD_LIB
getw	getc(3S)	BA_LIB
gmtime	ctime(3C)‡	BA_LIB
grep	grep(1)□	BU_CMD
group	group(4)	
grp	grp(5)	
gsignal	ssignal(3C)	BA_LIB
hcreate	hsearch(3C)	BA_LIB
hdestroy	hsearch(3C)	BA_LIB
hsearch	hsearch(3C)	BA_LIB
hypot	hypot(3M)†	BA_LIB
id	id(1)	AU_CMD
ioctl	ioctl(2)	BA_OS
ipc	ipc(2)●†	
ipc	ipc(5)●	
isalnum	ctype(3C)‡	BA_LIB
isalpha	ctype(3C)‡	BA_LIB
isascii	ctype(3C)‡	BA_LIB
isatty	ttyname(3C)	BA_LIB
iscntrl	ctype(3C)‡	BA_LIB
isdigit	ctype(3C)‡	BA_LIB
isgraph	ctype(3C)‡	BA_LIB
islower	ctype(3C)‡	BA_LIB
isprint	ctype(3C)‡	BA_LIB
ispunct	ctype(3C)‡	BA_LIB
isspace	ctype(3C)‡	BA_LIB
isupper	ctype(3C)‡	BA_LIB
isxdigit	ctype(3C)‡	BA_LIB
j0	bessel(3M)†	BA_LIB
j1	bessel(3M)†	BA_LIB
jn	bessel(3M)†	BA_LIB
join	join(1)	AU_CMD
rand48	drand48(3C)	BA_LIB
kill	kill(1)	BU_CMD
kill	kill(2)	BA_OS
l3tol	l3tol(3C)	
lcong48	drand48(3C)	BA_LIB

Interface	Entry	SVID
ld	ld(1D)	SD_CMD
ldexp	frexp(3C)	BA_LIB
lex	lex(1D)	SD_CMD
lfind	lsearch(3C)	BA_LIB
limits	limits(5)	
line	line(1)	BU_CMD
link	link(2)	BA_OS
lint	lint(1D)	SD_CMD
ln	cp(1)□	BU_CMD
localtime	ctime(3C)‡	BA_LIB
lock	lock(5)	
lockf	lockf(3C)	BA_OS
log	exp(3M)†	BA_LIB
log10	exp(3M)†	BA_LIB
logname	logname(1)	AU_CMD
logname	logname(3X)	
longjmp	setjmp(3C)	BA_LIB
lorder	lorder(1D)	SD_CMD
lp	lp(1)	AU_CMD
lpstat	lpstat(1)	AU_CMD
lrand48	drand48(3C)	BA_LIB
ls	ls(1)□	BU_CMD
lsearch	lsearch(3C)	BA_LIB
lseek	lseek(2)	BA_OS
l3tol	l3tol(3C)	
m4	m4(1D)	SD_CMD
mail	mail(1)	BU_CMD
mailx	mailx(1)†	AU_CMD
make	make(1D)	SD_CMD
mallinfo	malloc(3X)	BA_OS
malloc	malloc(3X)	BA_OS
malloc	malloc(5)	
mallopt	malloc(3X)	BA_OS
math	math(5)	
matherr	matherr(3M)†	BA_LIB
memccpy	memory(3C)	BA_LIB
memchr	memory(3C)	BA_LIB
memcmp	memory(3C)	BA_LIB
memcpy	memory(3C)	BA_LIB
memory	memory(5)	
memset	memory(3C)	BA_LIB
mesg	mesg(1)	AU_CMD



Interface	Entry	SVID
mkdir	mkdir(1) □	BU_CMD
mknod	mknod(1) †	AS_CMD
mknod	mknod(2)	BA_OS
mktemp	mktemp(3C)	BA_LIB
modf	frexp(3C)	BA_LIB
mon	mon(5)	
monitor	monitor(3C) †	SD_LIB
mount	mount(2)	BA_OS
rand48	drand48(3C)	BA_LIB
msg	msg(5) ●	
msgctl	msgctl(2) ● †	KE_OS
msgget	msgget(2) ● †	KE_OS
msgrcv	msgop(2) ● †	KE_OS
msgsnd	msgop(2) ● †	KE_OS
mv	cp(1) □	BU_CMD
newgrp	newgrp(1) †	AU_CMD
news	news(1) †	AU_CMD
nice	nice(2) †	KE_OS
nl	nl(1)	BU_CMD
nm	nm(1D)	SD_CMD
nohup	nohup(1)	BU_CMD
rand48	drand48(3C)	BA_LIB
null	null(7)	BA_ENV
od	od(1)	AU_CMD
open	open(2)	BA_OS
opendir	directory(3X)	
pack	pack(1)	BU_CMD
passwd	passwd(1)	AU_CMD
passwd	passwd(4)	
paste	paste(1)	BU_CMD
pause	pause(2)	BA_OS
pcat	pack(1)	BU_CMD
pclose	popen(3S)	BA_OS
perror	perror(3C)	BA_LIB
pg	pg(1)	BU_CMD
pipe	pipe(2)	BA_OS
plock	plock(2) †	KE_OS
popen	popen(3S)	BA_OS
pow	exp(3M) †	BA_LIB
pr	pr(1) □	BU_CMD
printf	printf(3S) ‡	BA_LIB
prof	prof(1D) †	SD_CMD

Interface	Entry	SVID
profil	profil(2) †	KE_OS
prs	prs(1D)	SD_CMD
ps	ps(1)	BU_CMD
ptrace	ptrace(2) †	KE_OS
putc	putc(3S)	BA_LIB
putchar	putc(3S)	BA_LIB
putenv	putenv(3C)	BA_LIB
putpwent	putpwent(3C)	SD_LIB
puts	puts(3S)	BA_LIB
pututline	getut(3C)	SD_LIB
putw	putc(3S)	BA_LIB
pwd	pwd(1)	BU_CMD
pwd	pwd(5)	BA_ENV
qsort	qsort(3C)	BA_LIB
rand	rand(3C)	BA_LIB
read	read(2)	BA_OS
readdir	directory(3X)	
realloc	malloc(3X)	BA_OS
red	ed(1) □	BU_CMD
regexp	regexp(3X)	BA_LIB
rewind	fseek(3S)	BA_OS
rewinddir	directory(3X)	
rm	rm(1) □	BU_CMD
rmdel	rmdel(1D)	SD_CMD
rmdir	rm(1) □	BU_CMD
HOME	environ(5)	BA_ENV
LOGNAME	environ(5)	BA_ENV
PATH	environ(5)	BA_ENV
TERM	environ(5)	BA_ENV
TZ	environ(5)	BA_ENV
sact	sact(1D)	SD_CMD
sbrk	brk(2) †	
scanf	scanf(3S) ‡	BA_LIB
sctfd	sct(7) †	
sctmt	sct(7) †	
sdb	sdb(1D) †	SD_CMD
search	search(5)	
sed	sed(1) □	BU_CMD
seed48	drand48(3C)	BA_LIB
seekdir	directory(3X)	
sem	sem(5) ●	
semctl	semctl(2) ● †	KE_OS

Interface	Entry	SVID
semget	semget (2)●†	KE_OS
semop	semop (2)●†	KE_OS
setbuf	setbuf (3C)	BA_LIB
setgid	setuid (2)	BA_OS
setgrent	getgrent (3C)	SD_LIB
setjmp	setjmp (3C)	BA_LIB
setjmp	setjmp (5)	
setkey	crypt (3C)	BA_LIB
setpgrp	setpgrp (2)	BA_OS
setpwent	getpwent (3C)	SD_LIB
setuid	setuid (2)	BA_OS
setutent	getut (3C)	SD_LIB
setvbuf	setbuf (3C)	BA_LIB
sh	sh (1)□	BU_CMD
shl	shl (1)†	AU_CMD
shm	shm (5)●	
shmat	shmop (2)●†	KE_OS
shmctl	shmctl (2)●†	KE_OS
shmdt	shmop (2)●†	KE_OS
shmget	shmget (2)●†	KE_OS
signal	signal (2)	BA_OS
signal	signal (5)	
signgam	gamma (3M)†	BA_LIB
sin	trig (3M)†	BA_LIB
sinh	sinh (3M)†	BA_LIB
size	size (1D)	SD_CMD
sleep	sleep (1)	BU_CMD
sleep	sleep (3C)	BA_OS
sort	sort (1)□	BU_CMD
spell	spell (1)	BU_CMD
split	split (1)	BU_CMD
sprintf	printf (3S)‡	BA_LIB
sqrt	exp (3M)†	BA_LIB
srand	rand (3C)	BA_LIB
srand48	drand48 (3C)	BA_LIB
sscanf	scanf (3S)‡	BA_LIB
ssignal	ssignal (3C)	BA_LIB
stat	stat (2)	BA_OS
stat	stat (5)	
stdio	stdio (3S)	
stdio	stdio (5)	
stime	stime (2)	BA_OS

Interface	Entry	SVID
strcat	string (3C)‡	BA_LIB
strchr	string (3C)‡	BA_LIB
strcmp	string (3C)‡	BA_LIB
strcpy	string (3C)‡	BA_LIB
strcspn	string (3C)‡	BA_LIB
string	string (5)	
strip	strip (1D)	SD_CMD
strlen	string (3C)‡	BA_LIB
strncat	string (3C)‡	BA_LIB
strncmp	string (3C)‡	BA_LIB
strncpy	string (3C)‡	BA_LIB
strpbrk	string (3C)‡	BA_LIB
strrchr	string (3C)‡	BA_LIB
strspn	string (3C)‡	BA_LIB
strtod	strtod (3C)‡	BA_LIB
strtok	string (3C)‡	BA_LIB
strtol	strtol (3C)	BA_LIB
stty	stty (1)	AU_CMD
su	su (1)	AU_CMD
sum	sum (1)	BU_CMD
swab	swab (3C)	BA_LIB
sync	sync (2)	BA_OS
sys_errlist	perror (3C)	BA_LIB
sys_nerr	perror (3C)	BA_LIB
system	system (3S)	BA_OS
tabs	tabs (1)	AU_CMD
tail	tail (1)	BU_CMD
tan	trig (3M)†	BA_LIB
tanh	sinh (3M)†	BA_LIB
tar	tar (1)	AU_CMD
tdelete	tsearch (3C)	BA_LIB
tee	tee (1)	BU_CMD
telldir	directory (3X)	
tempnam	tmpnam (3S)	BA_LIB
termio	termio (5)	
termio	termio (7)	BA_ENV
test	test (1)	BU_CMD
tfind	tsearch (3C)	BA_LIB
time	time (1D)	SD_CMD
time	time (2)	BA_OS
time	time (5)	
times	times (2)	BA_OS



Interface	Entry	SVID
times	times (5)	
timezone	ctime (3C)‡	BA_LIB
tmpfile	tmpfile (3S)	BA_LIB
tmpnam	tmpnam (3S)	BA_LIB
toascii	conv (3C)‡	BA_LIB
_tolower	conv (3C)‡	BA_LIB
tolower	conv (3C)‡	BA_LIB
touch	touch (1)	BU_CMD
_toupper	conv (3C)‡	BA_LIB
toupper	conv (3C)‡	BA_LIB
tr	tr (1)□	BU_CMD
true	true (1)	BU_CMD
tsearch	tsearch (3C)	BA_LIB
tsort	tsort (1D)	SD_CMD
tty	tty (1)	AU_CMD
tty	tty (7)	BA_ENV
ttyname	ttyname (3C)	BA_LIB
ttyslot	ttyslot (3C)	
twalk	tsearch (3C)	BA_LIB
types	types (5)	
tzname	ctime (3C)‡	BA_LIB
tzset	ctime (3C)‡	BA_LIB
ulimit	ulimit (2)	BA_OS
umask	umask (1)	BU_CMD
umask	umask (2)	BA_OS
umount	umount (2)	BA_OS
uname	uname (1)	BU_CMD
uname	uname (2)	BA_OS
unget	unget (1D)	SD_CMD
ungetc	ungetc (3S)	BA_LIB
uniq	uniq (1)	BU_CMD
unistd	unistd (5)	
unlink	unlink (2)	BA_OS
unpack	pack (1)	BU_CMD
ustat	ustat (2)	BA_OS

Interface	Entry	SVID
ustat	ustat (5)	
utime	utime (2)	BA_OS
utmp	utmp (4)	
utmp	utmp (5)	
utmpname	getut (3C)	SD_LIB
utsname	utsname (5)	
uucp	uucp (1)	AU_CMD
uulog	uucp (1)	AU_CMD
uuname	uucp (1)	AU_CMD
uupick	uuto (1)	AU_CMD
uustat	uustat (1)	AU_CMD
uuto	uuto (1)	AU_CMD
uux	uux (1)	AU_CMD
val	val (1D)	SD_CMD
values	values (5)	
varargs	varargs (5)	
vfprintf	vprintf (3S)	BA_LIB
vi	vi (1)	AU_CMD
vprintf	vprintf (3S)	BA_LIB
vsprintf	vprintf (3S)	BA_LIB
wait	wait (1)	BU_CMD
wait	wait (2)	BA_OS
wall	wall (1)	AU_CMD
wc	wc (1)□	BU_CMD
what	what (1D)	SD_CMD
who	who (1)	AU_CMD
write	write (1)	AU_CMD
write	write (2)	BA_OS
wtmp	utmp (4)	
xargs	xargs (1D)	SD_CMD
y0	bessel (3M)†	BA_LIB
y1	bessel (3M)†	BA_LIB
yacc	yacc (1D)	SD_CMD
yn	bessel (3M)†	BA_LIB

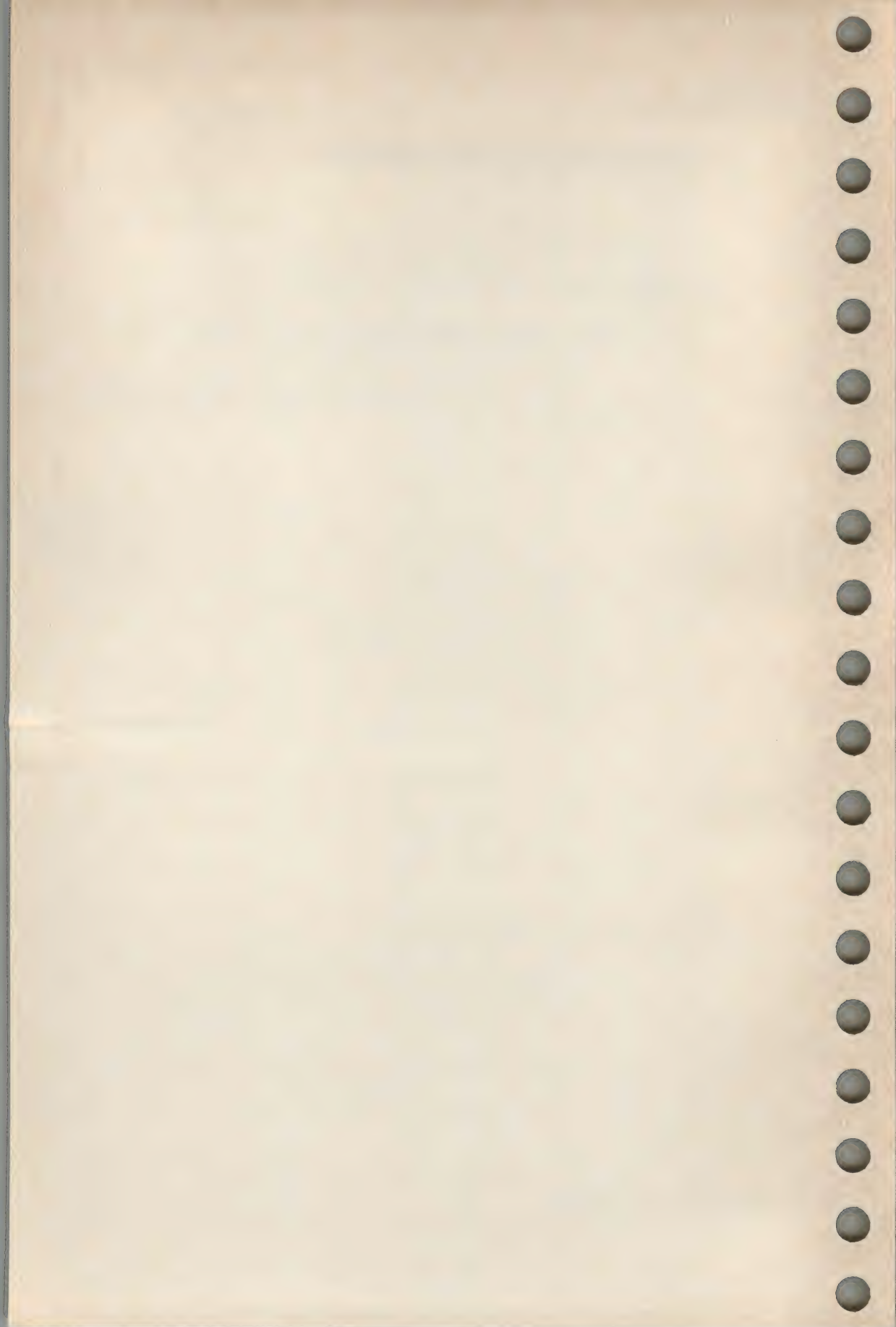
## **Commands and Utilities**

### **2.1 Commands and Utilities**

This Chapter contains the definitions of the Commands and Utilities. They fall into two categories

1. Standard Utilities, which are present on every X/OPEN system. These are indicated by a suffix of (1) after their name, for example *ar*(1).
2. Software Development utilities, which are present only on systems supporting the software development facility. These are indicated by the suffix (1D), for example *yacc*(1D).





## NAME

*admin* — create and administer SCCS files

## SYNOPSIS

```
admin [—n] [—i[name]] [—rrel] [—t[name]] [—fflag[flag-val]]  
      [—dflag[flag-val]] [—alogin] [—eloglein] [—m[mrlist]] [—y[comment]]  
      [—h] [—z] file ...
```

## DESCRIPTION

The command *admin* is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of options, which begin with —, and named *files* (note that SCCS file names must begin with *s.*). If a named *file* does not exist, it is created, and its parameters are initialised according to the specified options. Parameters not initialised by an option are assigned a default value. If a named *file* does exist, parameters corresponding to specified options are changed, and other parameters are left as is.

If *file* is a directory, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with *s.*) and unreadable files are silently ignored. If a name of — is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The options are as follows. Each is explained as though only one named file is to be processed since the effects of the options apply independently to each named file.

—n This option indicates that a new SCCS file is to be created.

—i[*name*]

The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see —r option for delta numbering scheme). If the —i option is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this option is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an *admin* command on which the —i option is supplied. Using a single *admin* to create two or more SCCS files requires that they be created empty (no —i option). Note that the —i option implies the —n option.

—rrel The *release* into which the initial delta is inserted. This option may be used only if the —i option is also used. If the —r option is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).



**—t[*name*]**

The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **—t** option is used and *admin* is creating a new SCCS file (the **—n** and/or **—i** options also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **—t** option without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **—t** option with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

**—f*flag***

This option specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **—f** options may be supplied on a single *admin* command line. The allowable *flags* and their values are:

**b** Allows use of the **—b** option on a *get* command to create branch deltas.

**cceil** The highest release (i.e., "*ceiling*"), a number less than or equal to 9999, which may be retrieved by a *get* command for editing. The default value for an unspecified *c* flag is 9999.

**ffloor** The lowest release (i.e., "*floor*"), a number greater than 0 but less than 9999, which may be retrieved by a *get* command for editing. The default value for an unspecified *f* flag is 1.

**dSID** The default delta number (SID) to be used by a *get* command.

**i[*str*]** Causes the message issued by *get* or *delta* to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(1D)) are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly match the given string; however, the string must contain a keyword, and no embedded newlines.

**j** Allows concurrent *get* commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

**l*list*** A *list* of releases to which deltas can no longer be made (*get* **—e** against one of these "locked" releases fails). The *list* has the following syntax:

**<list> ::= <range> | <list> , <range>**

**<range> ::= RELEASE NUMBER | a**

The character *a* in the *list* is equivalent to specifying all releases for the named SCCS file.

**n** Causes *delta* to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a new release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

**qtext** User definable *text* substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*.

**mmod**

Module name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*. If the *m* flag is not specified, the value assigned is the name of the SCCS file with the leading *s*. removed.

**ttype** Type of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*.

**v[pgm]**

Causes *delta* to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program. (If this flag is set when creating an SCCS file, the *m* option must also be used even if its value is null.)

**—dflag**

Causes removal (deletion) of the specified *flag* from an SCCS file. The **—d** option may be specified only when processing existing SCCS files. Several **—d** options may be supplied on a single *admin* command. See the **—f** option for allowable *flag* names. (The *l*list flag gives a *list* of releases to be "unlocked". See the **—f** option for further description of the *l* flag and the syntax of a *list*.)

**—alogin**

A *login* name, or numerical group ID, to be added to the list of users who may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several *a* options may be used on a single *admin* command line. As many *logins*, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a *!* the users so specified are denied permission to make deltas.

**—elogin**

A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **—e** options may be used on a single *admin* command line.



**—y[comment]**

The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*. Omission of the **—y** option results in a default comment line being inserted in the form:

date and time created YY/MM/DD HH:MM:SS by login

The **—y** option is valid only if the **—i** and/or **—n** options are specified (i.e., a new SCCS file is being created).

**—m[mrlist]**

The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*. The *v* flag must be set and the MR numbers are validated if the *v* flag has a value (the name of an MR number validation program). Diagnostics will occur if the *v* flag is not set or MR validation fails.

**—h**

Causes *admin* to check the structure of the SCCS file, and to compare a newly computed check sum (the sum of all the characters in the SCCS file except those in the first line) with the check sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This option inhibits writing on the file, so that it nullifies the effect of any other options supplied. It is only meaningful when processing existing files.

**—z**

The SCCS file checksum is recomputed and stored in the first line of the SCCS file (see **—h**, above).

Note that use of this option on a truly corrupted file may prevent future detection of the corruption.

## FILES

All SCCS file names must be of the form *s.file-name*. New SCCS files are given read-only permission mode (see *chmod*(1)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called *x.file-name*, see *get*(1D), created with read-only mode if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

The command *admin* also makes use of a transient lock file (called *z.file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1D) for further information.

## SEE ALSO

*delta*(1D), *get*(1D), *prs*(1D), *what*(1D).

## APPLICATION USAGE

It is recommended that directories containing SCCS files be writable by the owner only, and that SCCS files themselves be read-only. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

CHANGE HISTORY

First released in Issue 2.

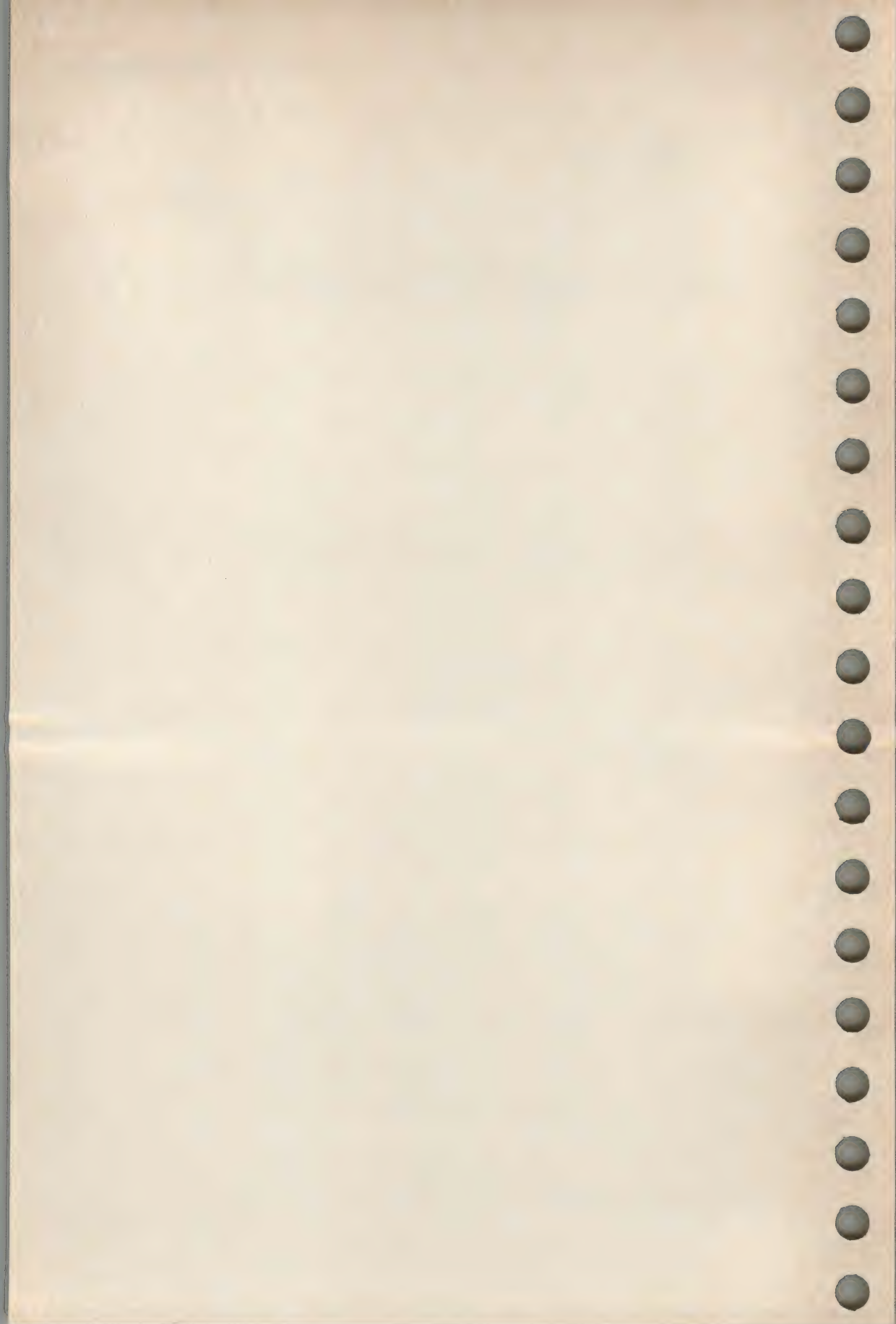
Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The SYNOPSIS has had the —% option corrected to the —y option.

The last sentence of the description of the *i[str]* flag has been added from the System V Release 2.0 manual page.





## NAME

ar — archive and library maintainer

## SYNOPSIS

ar option [posname] afile [name ...]

## DESCRIPTION

The *ar* command maintains groups of files combined into a single archive file. It is used to create and update library files as used by the link editor, see *ld*(1D). It can be used, however, for any similar purpose. If an archive file is created from printable files, the entire archive file is printable.

When *ar* creates an archive file, it creates administrative information in a format that is portable across all machines. When there is at least one object file (that *ar* recognises as such) in the archive, an archive symbol table is created in the archive file and maintained by *ar*. The archive symbol table is never visible or accessible to the user. (It is used by the link editor to search the archive file.) Whenever the *ar* command is used to create or update the contents of such an archive, the symbol table is rebuilt. The *s* modifier character described below forces the symbol table to be rebuilt.

The argument *option* is a — followed by one character from the set *drqtpmx* which may be optionally concatenated with one or more characters to modify the action. These modifier characters are taken from the set *vuabicl*s but not all modifiers make sense with all options. See below for further explanation. The argument *posname* is the name of a file in the archive file, used for relative positioning; see options —*r* and —*m* below. The argument *afile* is the archive file. The *names* are constituent files in the archive file.

The meanings of the *option* characters are:

- d* Delete the named files from the archive file. Valid modifiers are *vl*.
- r* Replace the named files in the archive file. Valid modifiers are *vuabicl*. If the modifier *u* is used, then only those files with dates of modification later than the archive files are replaced. If an optional positioning character from the set *abi* is used, then the *posname* argument must be present, and specifies that new files are to be placed after (*a*) or before (*b* or *i*) *posname*. Otherwise new files are placed at the end.
- q* Quickly append the named files to the end of the archive file. Valid modifiers are *vcl*. In this case *ar* does not check whether the added members are already in the archive. This is useful to bypass the searching otherwise done when creating a large archive piece-by-piece.
- t* Print a table of contents of the archive file. If no names are given, all files in the archive are listed. If names are given, only those files are listed. Valid modifiers are *vs*. The *v* modifier gives a long listing of all information about the files.
- p* Print the named files from the archive. Valid modifiers are *vs*.



- m Move the named files to the end of the archive. Valid modifiers are *vabil*. If a positioning modifier from the set *abi* is present, then the *posname* argument must be present and, as with the option *r*, it specifies where the files are to be moved.
- x Extract the named files. If no names are given, all files in the archive are extracted. The archive file is not changed. Valid modifiers are *vs*.

The meanings of the modifier characters are:

- v Give verbose output. When used with the option characters *d*, *r*, *q*, or *m* this gives a verbose file-by-file description of the making of a new archive file from the old archive (if one exists) and the constituent files. When used with *x*, this precedes each file with its name.
- c Suppress the message that is produced by default when the archive file *afile* is created.
- l Place temporary files in the local current working directory, rather than in the directory specified by the environment variable *TMPDIR* or in the default directory.

UN

**s** Force the regeneration of the archive symbol table even if *ar* is not invoked with a command which will modify the archive file contents. This command is useful to restore the archive symbol table after it has been stripped, see *strip(1D)*.

#### SEE ALSO

*ld(1D)*, *strip(1D)*, *lorder(1D)*.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The sentence on the portability of archives has been removed.

## NAME

as — common assembler (OPTIONAL)

## SYNOPSIS

as [—o *objfile*] [—m] [—V] *file*

## DESCRIPTION

The *as* command assembles the named *file*. The following options may be specified in any order:

—o *objfile*

Put the output of the assembly in *objfile*. By default, the output file name is formed by removing the *.s* suffix, if there is one, from the input file name and appending a *.o* suffix.

MV UN —m Run the *m4* macro pre-processor on the input to the assembler.

MV PI —V Write the version number of the assembler being run on the standard error output.

## SEE ALSO

cc(1D), ld(1D), m4(1D).

## APPLICATION USAGE

The command *cc* is the recommended interface to the assembler. The *as* command itself may not be present on all X/OPEN systems.

## FUTURE DIRECTIONS

Users will be able to specify, by means of the *TMPDIR* environment variable, the directory in which any temporary files are to be created.

This addition is part of the effort to eliminate hard-coded pathnames from the compilation system.

## CHANGE HISTORY

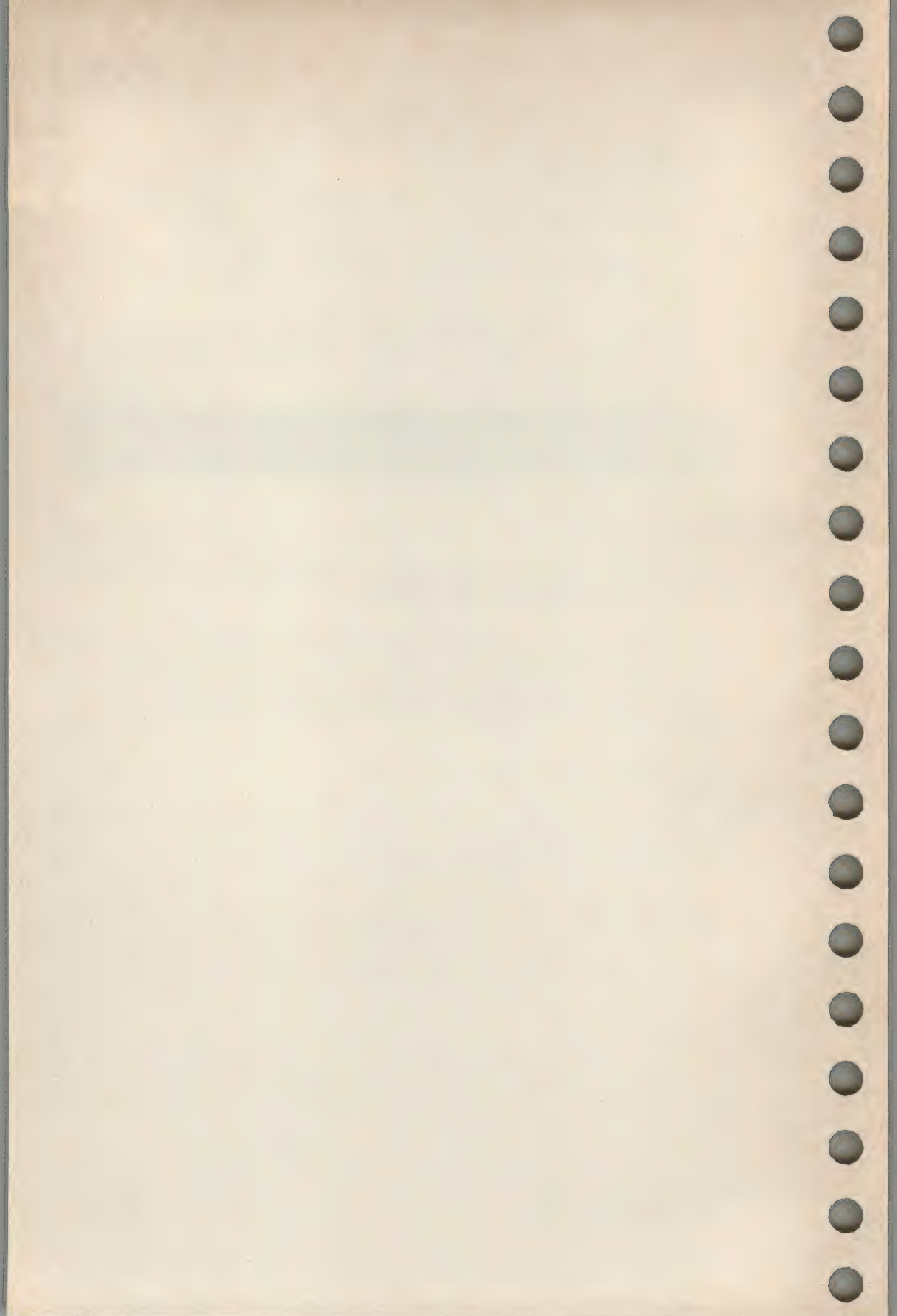
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The references to *.s* and *.o* have been added in the description of the —o flag.





## NAME

*at*, *batch* — execute commands at a later time

## SYNOPSIS

*at* time [date] [+ increment]

*at* -r job ...

*at* -l [job ...]

*batch*

## DESCRIPTION

The commands *at* and *batch* read commands from standard input to be executed at a later time. The command *at* allows the user to specify when the commands should be executed, while jobs queued with *batch* will execute when system load level permits.

*at*

## OF

The option -r to *at* removes jobs previously scheduled by *at* or *batch*. The job number is the number reported at invocation by *at* or *batch*. Job numbers can also be obtained by using the -l option. Only the super-user is allowed to remove another user's jobs.

Standard output and standard error output are mailed to the user unless they are redirected. The environment variables, current directory, *umask*, and *ulimit* are retained when the commands are executed. Open files, traps, and priority are lost.

Users are permitted to use *at* if their name appears in the file */usr/lib/cron/at.allow*. If that file does not exist, the file */usr/lib/cron/at.deny* is checked to determine if the user should be denied access to *at*. If neither file exists, only the super-user is allowed to submit a job. If only *at.deny* exists and is empty, global usage is permitted. The allow/deny files consist of one user name per line.

The *time* may be specified as 1, 2, or 4 digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning *hour:minute*. A suffix *am* or *pm* may be appended; otherwise a 24-hour clock time is understood. The suffix *zulu* may be used to indicate GMT. The special names *noon*, *midnight*, *now*, and *next* are also recognised.

An optional *date* may be specified as either a month name followed by a day number (and possibly year number preceded by a comma) or a day of the week (fully spelled or abbreviated to three characters). Two special "days", *today* and *tomorrow*, are recognised. If no *date* is given, *today* is assumed if the given hour is greater than the current hour and *tomorrow* is assumed if it is less. If the given month is less than the current month (and no year is given), next year is assumed.

The optional *increment* is a number suffixed by one of the following: *minutes*, *hours*, *days*, *weeks*, *months*, or *years*. (The singular form is also accepted.)



Thus legitimate commands include:

```
at 0815am Jan 24
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday
```

OF

The commands *at* and *batch* write the job number and schedule time to standard error.

### batch

The command *batch* submits a batch job. It is almost equivalent to *at now*, but not quite. For one, the job goes into a different queue. For another, *at now* does not work: it is too late (and results in an error message).

### EXAMPLES

The *at* and *batch* commands read from standard input the commands to be executed at a later time. It may be useful to redirect standard output within the specified commands.

1. This sequence can be used at a terminal:

```
batch
spell filename >outfile
<EOF>
```

2. This sequence, which demonstrates redirecting standard error to a pipe, is useful in a command procedure (the sequence of output redirection specifications is significant):

```
batch <<!
spell filename 2>&1 >outfile | mail loginid
!
```

3. To have a job reschedule itself, *at* can be invoked from within the shell procedure, by including code similar to the following within the shell file:

```
echo "sh shellfile" | at 1900 thursday next week
```

### FILES

/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users

### SEE ALSO

crontab(1).

### APPLICATION USAGE

Commands will be executed using *sh*(1).

### CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

awk — pattern-directed scanning and processing language

## SYNOPSIS

```
awk [—Fc] program [parameters] [file...]  
awk [—Fc] —f progfile [parameters] [file...]
```

## DESCRIPTION

The *awk* command executes programs written in the *awk* programming language, which is specialised for data manipulation. An *awk* program is a sequence of patterns and corresponding actions. When input is read that matches a pattern, the action associated with that pattern is carried out.

The *file* arguments contain the input to be read. If no files are given or the filename — is given, the standard input is used.

Each line of input is matched in turn against the set of patterns in the program. The *awk* program may either be in a file *progfile* or may be specified in the command line, taking into account the quoting rules of the command interpreter.

Each line of input is matched in turn against each pattern in the program. For each pattern matched, the associated action is executed.

The *awk* command interprets each input line as a sequence of fields where, by default, a field is a string of non-blank, non-tab characters. This default whitespace field delimiter can be changed by using the *—Fc* option, or the variable *FS* see below. The command *awk* denotes the first field in a line *\$1*, *\$2*, and so forth. *\$0* refers to the entire line. Setting any other field causes the re-evaluation of *\$0*.

Pattern-action statements in an *awk* program have the form:

```
pattern { action }
```

In any pattern-action statement, either the pattern or the action may be omitted. A missing action means print the input line to the standard output; a missing pattern is always matched, and its associated action is executed for every input line read.

## Patterns

Patterns are *special patterns* or arbitrary Boolean combinations (*!*, *||*, *&&*, and parentheses) of regular expressions and relational expressions. The operator *!* has the highest precedence, then *&&* and then *||*. Evaluation is left to right and stops when truth or falsehood has been determined.

Boolean Operator	Meaning
!	negation
&&	and
	or



### Special Patterns

The *awk* command recognises two special patterns, *BEGIN* and *END*. *BEGIN* is matched once and its associated action executed before the first line of input is read. *END* is matched once and its associated action executed after the last line of input has been read. (See the fifth and eighth examples.) These two patterns must have associated actions.

### Relational Expressions

A pattern may be any expression that compares strings of characters or numbers. A relational expression is either

expression relational-operator expression

or

expression matching-operator regular-expression

The six relational operators are shown in the table below; regular expression matching operators are described later. In a comparison, if both operands are numeric, a numeric comparison is made, otherwise, a string comparison is made.

Relational Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
!=	not equal to
==	equal to

### Regular Expressions

*Awk* makes use of extended regular expression syntax.

A regular expression must be surrounded by slashes. If *re* is a regular expression, then the pattern

*/re/*

matches any line of input that contains a substring specified by the regular expression. A regular expression comparison may be limited to a specific field by one of the two regular expression matching operators, *~* and *!~*.

*\$4 ~ /re/*

matches any line in which the 4th field matches the regular expression */re/*.

*\$4 !~ /re/*

matches any line in which the 4th field does not match the regular expression */re/*.

**Pattern Ranges**

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the following occurrence of the second pattern.

**Variables and Special Variables**

Variables may be used in an *awk* program by assigning to them. They do not need to be declared. Like *field* variables, all variables are treated as string variables unless used in a clearly numeric context (see **Relational Expressions**). Field variables are designated by a *\$* followed by a number or numerical expression. New field variables may be created by assigning a value to them. If not initialised, variables default to the null string or zero. Other special variables set by *awk* are shown in the table below.

Special Variable	Meaning
\$n	The string read as field n.
FS	Input field separator. Set to whitespace by default.
RS	Input record separator. Set to newline by default.
FILENAME	Name of the current input file.
NF	Number of fields in the current record.
NR	Ordinal number of the current record from the start of input.
OFMT	<i>Print</i> statement output format for numbers. <i>%.6g</i> by default.
OFS	<i>Print</i> statement output field separation. One blank by default.
ORS	<i>Print</i> statement output record separator. Newline by default.

Variable assignments of the form *x=value* may occur on the command line as *parameters*.

**Actions**

An action is a sequence of statements. A statement can be one of the following. Square brackets indicate optional elements.



```

if ( expression ) statement [ else statement ]
while ( expression ) statement
for ( expression ; expression ; expression ) statement
for ( variable in array ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next
exit ( expression )

```

Any single statement may be replaced by a statement list enclosed in curly braces. The statements in a statement list are separated by newlines or semicolons. The symbol `#` anywhere in a program line begins a comment, which is terminated by the end of the line.

Statements are terminated by semicolons, newlines, or right braces. A long statement may be split across several lines by ending each partial line with a `\`. An empty expression-list stands for the whole input line. Expressions take on string or numeric values as appropriate, and are built using the operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus operator), and concatenation (indicated by a blank between strings in an expression). The C language operators `++`, `--`, `+=`, `-=`, `*=`, `/=`, and `%=` are also available in expressions. Variables may be scalars, array elements (denoted `x[i]`) or fields. Variables are initialised to the null string. Array subscripts may be any string, not necessarily numeric.

String constants are surrounded by double quotes (`"..."`). A string expression is created by concatenating constants, variables, field names, array elements, functions and other expressions.

The *expression* acting as the conditional in an *if* statement can include the relational operators, the regular expression matching operators, logical operators, juxtaposition for concatenation and parentheses for grouping. *Expression* is evaluated and if it is non-zero and non-null, *statement* is executed; otherwise, if *else* is present, the *statement* following the *else* is executed.

The *while*, *for*, *break*, and *continue* statements are as in the C language.

The *print* statement prints its arguments on the standard output separated by the current output field separator (see variable *OFS* above) and terminated by the output record separator (see variable *ORS*, above). The *printf* statement formats its expression list according to *format*, see *printf*(3S). Output can be redirected to a file if *>expr* is present, appended (*>>expr*), or sent to a pipe (*|expr*).

The *next* statement causes the next input line to be scanned, skipping the remaining characters on the current input line. The *exit* statement causes the termination of the *awk* program, skipping the rest of the input.

## Built-In Functions

The built-in function *length(s)* returns the length of its arguments taken as a string, or of the whole line, *\$0*, if there is no argument.

There are also built-in functions *exp(x)* (the exponential function of *x*), *log(x)* (natural logarithm of *x*), *sqrt(x)* (square root of *x*), and *int(x)* (truncates its argument to an integer).

*Split(s1, arr, s2)* assigns the fields of the string *s1* to successive elements of the array *arr*, using the characters in string *s2* as field separators. Assignments start at array element *arr[1]*. *Substr(s, p, n)* returns the at most *n*-character substring of *s* that begins at position *p*. *Index(s1, s2)* returns the leftmost position where the string *s2* occurs in *s1* or zero if *s2* does not occur in *s1*.

The *getline* function immediately reads the next input record. Fields *NR* and *\$0* are all set, but control is left at exactly the same spot in the *awk* program. *Getline* returns 0 on end of file, and 1 otherwise.

The function *sprintf(fmt, expr, expr, ...)* formats the expressions according to the *printf(3S)* format given by *fmt* and returns the resulting string.

## EXAMPLES

The following are examples of simple *awk* programs:

1. Print on the standard output all input lines for which field 3 is greater than 5.

```
$3 > 5
```

2. Print every 10th line.

```
(NR % 10) == 0
```

3. Print any line with a substring matching the regular expression.

```
/(G|D)(2[0-9][a-zA-Z]*)/
```

4. Print the second to the last and the last field in each line. Separate the fields by a colon.

```
{OFS=":";print $(NF-1), $NF}
```

5. Print the line number and number of fields in each line. The 3 strings representing the line number, the colon and the number of fields are concatenated and that string is printed.

```
BEGIN {line = 0}
{line = line + 1
 print line ":" NF}
```

6. Print lines longer than 72 characters.

```
length > 72
```

7. Print the first two fields in opposite order separated by the *OFS*.

```
{ print $2, $1 }
```



8. Add up the first column, print sum and average.

```
{s += $1 }
END {print "sum is ", s, " average is", s/NR}
```

9. Print fields in reverse order.

```
{ for (i = NF; i > 0; --i) print $i }
```

10. Print all lines between occurrences of the strings *start* and *stop*.

```
/start/, /stop/
```

11. Print the all lines in which the first field differs that of the previous line.

```
$1 != prev { print; prev = $1 }
```

12. Print file, filling in page number starting at 5.

```
/Page/ { $2 = n++; }
{ print }
```

13. Command line.

```
awk -f program n=5 input
```

## APPLICATION USAGE

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number 0 should be added to it; to force it to be treated as a string concatenate the null string ( " " ) to it.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

In the third paragraph, the words "as a string enclosed in single quotes" have been replaced by "", taking into account the quoting rules of the command interpreter".

In the description of the match operator, the examples now show the ~ and !~ operators correctly.

The following have been added from the System V Release 2.0 Support Tools Guide:

- The description of command line variable assignments.
- The fact that variable values default to the null string or zero.
- The description of the RS special variable.
- The descriptions of appending to a file, and of output to a pipe.
- The descriptions of the *split()*, *index()* and *getline* functions.
- The description of *for ( variable in array ) statement*

The complete description of regular expressions has been replaced by the paragraph "Awk makes use of extended regular expression syntax."

## NAME

banner — make large letters

## SYNOPSIS

banner string...

OF

## DESCRIPTION

The command *banner* writes each argument *string* in large letters (across the page) to the standard output, putting each argument on a separate "line". Spaces can be included in an argument by surrounding it with quotes. The maximum number of characters that can be accommodated in a line is implementation dependent, however, a minimum of 10 can be assumed; excess characters are simply ignored.

## APPLICATION USAGE

The font of the output is that of the US ASCII character set.

## SEE ALSO

echo(1).

## CHANGE HISTORY

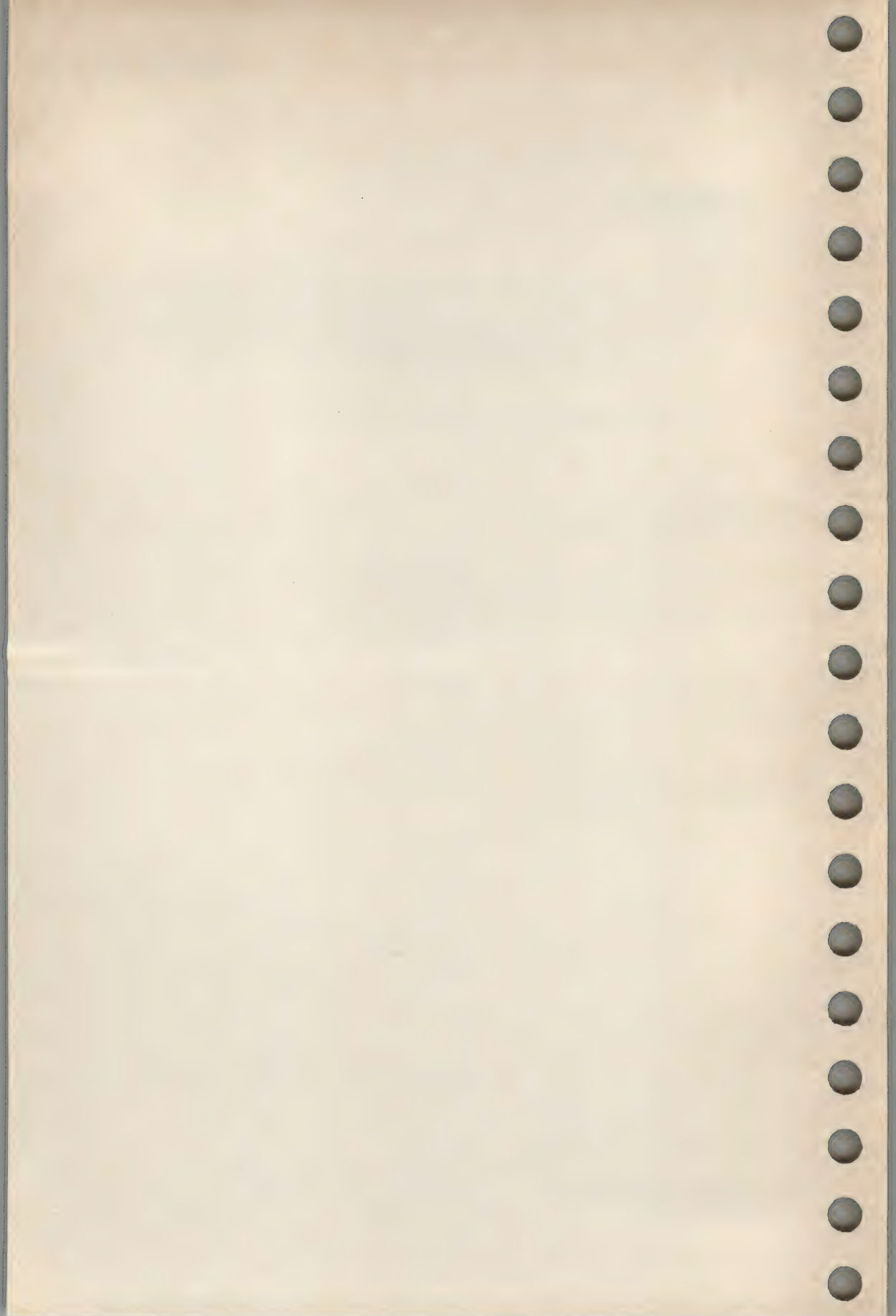
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The specified minimum of 10 characters has been added.





## NAME

basename, dirname — deliver portions of path names

## SYNOPSIS

basename string [ suffix ]

dirname string

## DESCRIPTION

## basename

The command *basename* deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and writes the result to the standard output. It is normally used inside substitution marks (``) within command procedures.

## dirname

The command *dirname* delivers all but the last level of the path name given in *string*.

## EXAMPLES

1. The following example moves the file named *abc* to a file named *xyz* in the current directory:

```
mv abc `basename /p/q/xyz.c .c`
```

2. The following example will set the variable *NAME* to */usr/src/cmd*:

```
NAME=`dirname /usr/src/cmd/xyz.c`
```

## SEE ALSO

sh(1).

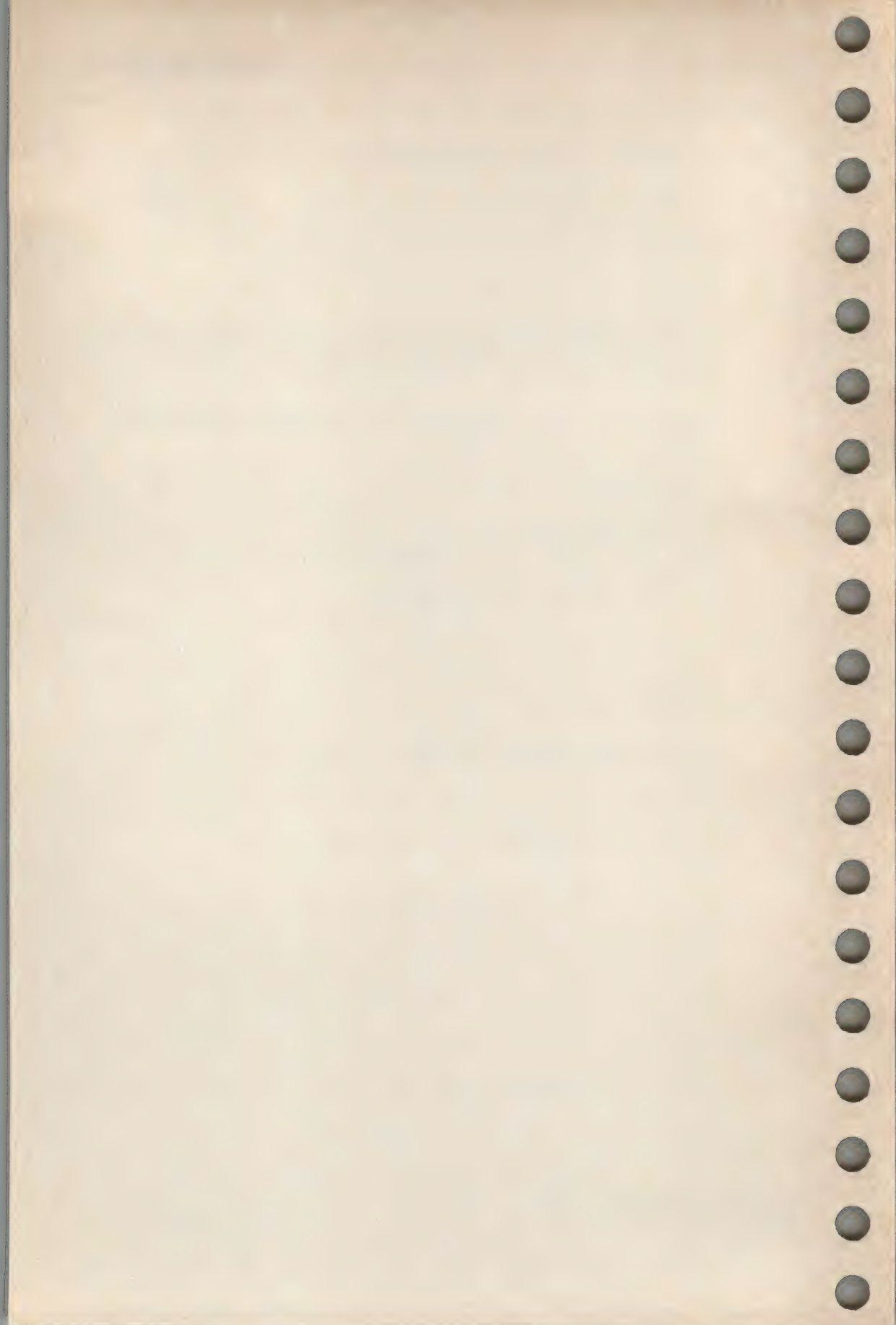
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

cal — print calendar

## SYNOPSIS

cal [ *[ month ] year* ]

LA OF

## DESCRIPTION

The *cal* command writes a calendar for the specified *year* on the standard output. If *month* is also specified, a calendar for that month only is printed. If neither is specified, a calendar for the present month is printed. The argument *year* can be between 1 and 9999. (Note that *cal 83* refers to 83 A.D., not 1983.) The *month* is a number between 1 and 12.

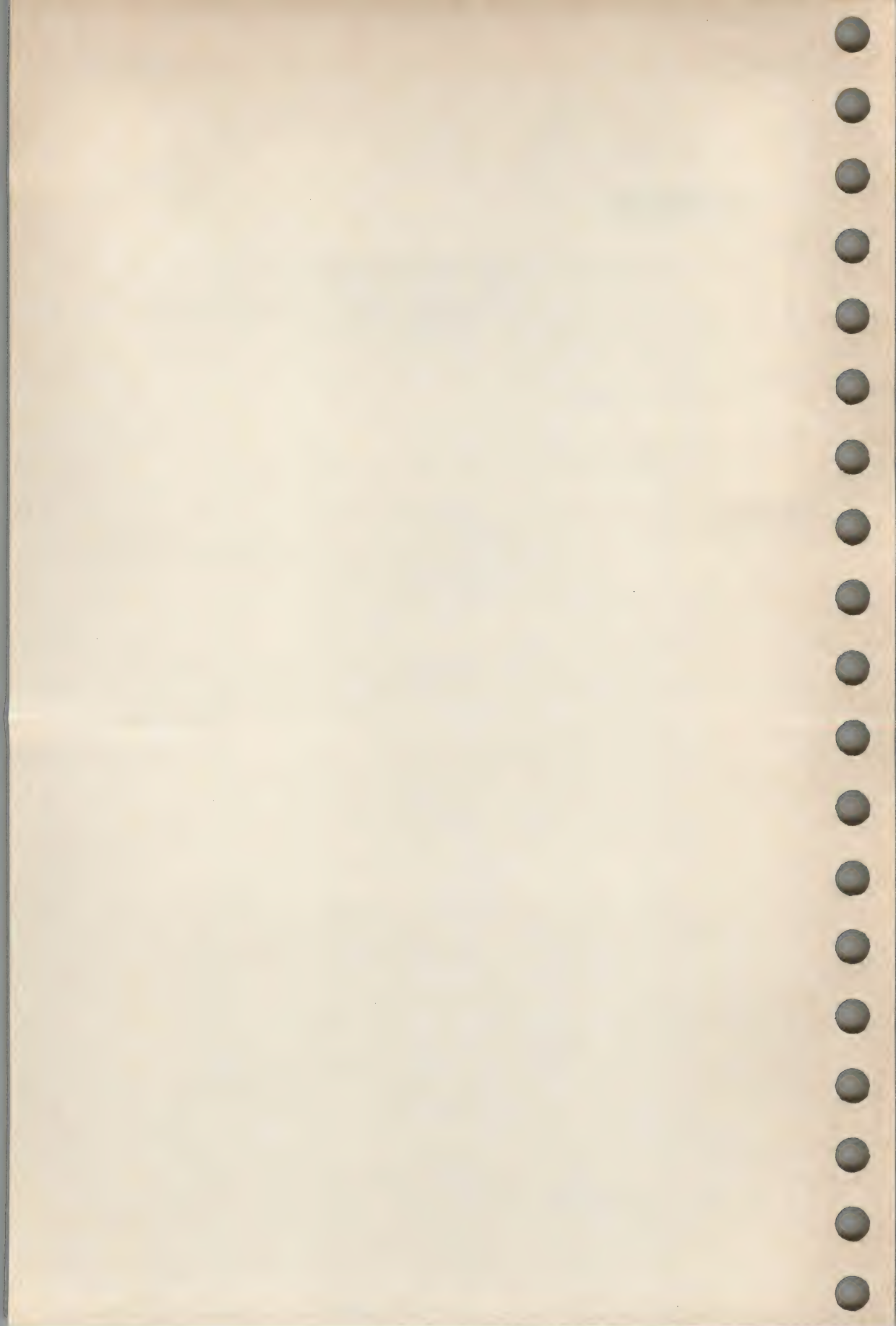
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





NAME

calendar — reminder service

SYNOPSIS

calendar

DESCRIPTION

The command *calendar* consults the file *calendar* in the current directory and writes lines that contain today's or tomorrow's date anywhere in the line on the standard output. Month-day date formats such as *Aug. 24*, *august24* and *8/24* are recognised. On weekends, *tomorrow* extends through Monday.

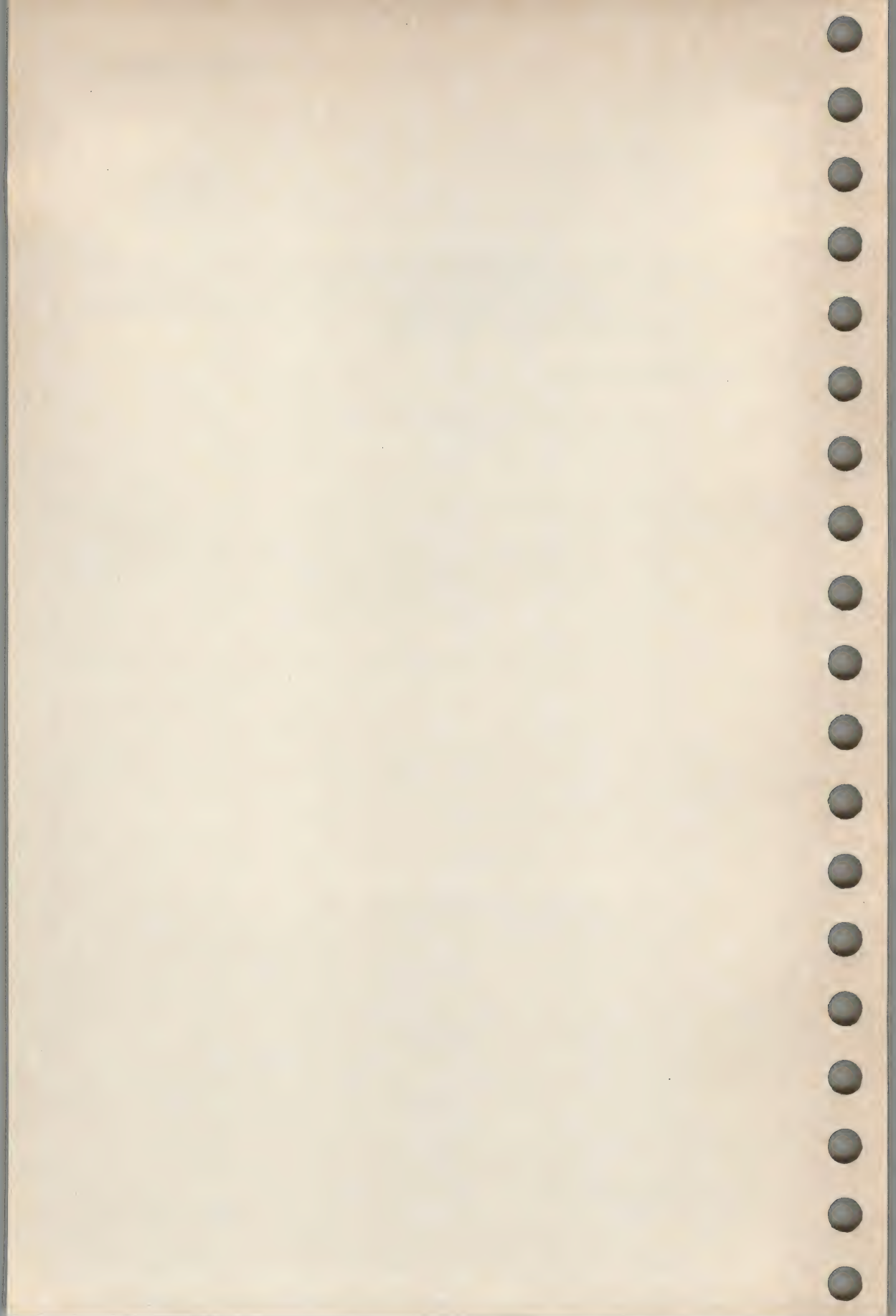
CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

cat — concatenate and print files

## SYNOPSIS

cat [**—s**] [*file*...]

## DESCRIPTION

The command *cat* reads each *file* in sequence and writes it to the standard output. Thus:

cat file

writes the file on the standard output, and:

cat file1 file2 >file3

concatenates the first two files and writes the result to the third.

If no input file is given, or if the argument **—** is encountered, *cat* reads from the standard input.

PI

The **—s** option makes *cat* silent about non-existent files.

## APPLICATION USAGE

Because of the mechanism used to perform output redirection, a command such as this:

cat file1 file2 >file1

will cause the original data in *file1* to be lost.

## CHANGE HISTORY

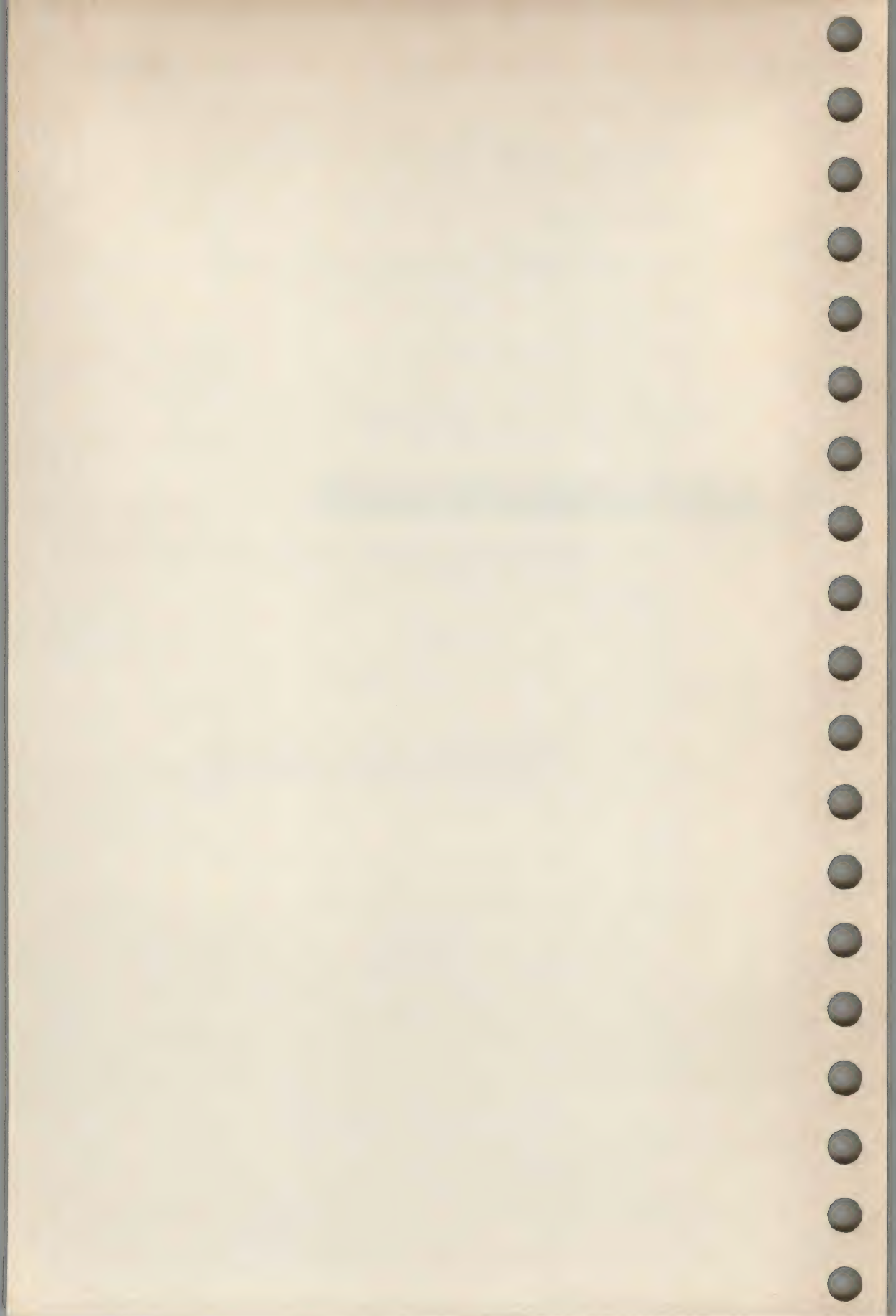
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The SYNOPSIS has been modified to indicate that the *file* arguments are optional.





## NAME

cc — C compiler

## SYNOPSIS

cc [ options ] file ...

## DESCRIPTION

The `cc` command is the interface to the C compilation system. The system conceptually consists of a preprocessor, compiler, optimiser, assembler and link editor. The `cc` command processes the supplied *options* and then executes the various tools with the appropriate arguments.

The suffix of *file* indicates how it is to be treated. Files whose names end with `.c` are taken to be C source programs, and may be preprocessed, compiled, optimised, and link edited. The compilation process may be stopped after the completion of any pass if the appropriate *options* are supplied. If the compilation process is allowed to complete the assembly phase, then an object program is produced; the object program for a source file called `xyz.c` is created in a file called `xyz.o`. However, the `.o` file is normally deleted if a single C program is compiled and link edited all at one go.

In the same way, arguments whose names end with `.s` are taken to be assembly source programs, and may be assembled and link edited. Files with names ending in `.i` are taken to be preprocessed C source programs and may be compiled, optimised, assembled, and link edited. Files whose names do not end in `.c`, `.s`, or `.i` are handed to the link editor.

By default, if an executable file is produced (i.e., the link edit phase is allowed to complete) the file is called `a.out`. This default name can be changed with the `—o` option (see below).

The following options are interpreted by `cc`:

- `—c` Suppress the link edit phase of the compilation, and do not remove any object files that are produced.
- `—f` Include floating-point support for systems without an automatically included floating-point implementation. This option is ignored on systems that do not need it.

PI OP

- `—g` Cause the compiler to generate additional information needed for use by `sdb(1D)`.

- `—o outfile`

Use the name *outfile* instead of the default `a.out` for the executable file produced. This is a link editor option.

PI OP

- `—p` Arrange for the compiler to produce code that counts the number of times each routine is called; also, if link editing takes place, a profiled version of the standard C library is linked and *monitor*, see `monitor(3C)`, is automatically called. A `mon.out` file will then be produced at normal termination of execution of the program. An execution profile can then be generated by use of `prof(1D)`.



- PI —**q** This option is reserved for specification of implementation specific profiling directives.
- UN —**E** Run only the preprocessor on the named C programs and send the result to the standard output.
- PI —**F** This option is reserved for implementation specific optimisation directives.
- O** Do compilation phase optimisation. This option will not affect *.s* files.
- UN —**P** Run only the preprocessor on the named C programs and leave the result on corresponding files suffixed *.i*.
- UN —**S** Compile and do not assemble the named C programs, and leave the assembler-language output on corresponding files suffixed *.s*.
- PI —**Wc, arg[, arg ...]**  
Hand off the argument[s] *arg* to phase *c* where *c* is one of [*p02aI*] indicating preprocessor, compiler, optimiser, assembler or link editor, respectively. For example, **—Wa**, **—m** passes **—m** to the assembler phase.

The *cc* command also recognises the options **—C**, **—D**, **—I**, and **—U**, and passes them (and their associated arguments) directly to the preprocessor without using the **—W** option. Similarly, the link editor options **—e**, **—l**, **—o**, **—r**, **—s**, **—u**, **—L**, and **—V** are recognised and passed directly to the link editor. See *cpp(1D)* and *ld(1D)* for descriptions of these options.

Other arguments are taken to be C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines, and are passed directly to the link editor. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name *a.out* (unless the **—o** link editor option is used).

The standard C library is automatically available to the C program. Other libraries (including the math library) must be specified explicitly using the **—l** option with *cc*; see *ld(1D)* for details.

#### FILES

<i>file.c</i>	input file
<i>file.i</i>	preprocessed C source file
<i>file.o</i>	object file
<i>file.s</i>	assembly language file
<i>a.out</i>	link edited (executable) output

#### SEE ALSO

*cpp(1D)*, *ld(1D)*, *prof(1D)*, *sdb(1D)*, *exit(2)*, *monitor(3C)*.

#### APPLICATION USAGE

Arbitrary length variable names are allowed in the C language, starting with System V Release 2.0.

Since the *cc* command usually creates files in the current directory during the compilation process, it is typically necessary to run the *cc* command in a directory in which a file can be created.

#### FUTURE DIRECTIONS

The SVID reserves the `—Y` option for future use. It will be used to allow the user to specify the directories searched by the various components of `cc`.

Users will also be able to specify, by means of the `TMPDIR` environment variable, the directory in which any temporary files are to be created.

These additions are part of the effort to eliminate hard-coded pathnames from the compilation system.

#### CHANGE HISTORY

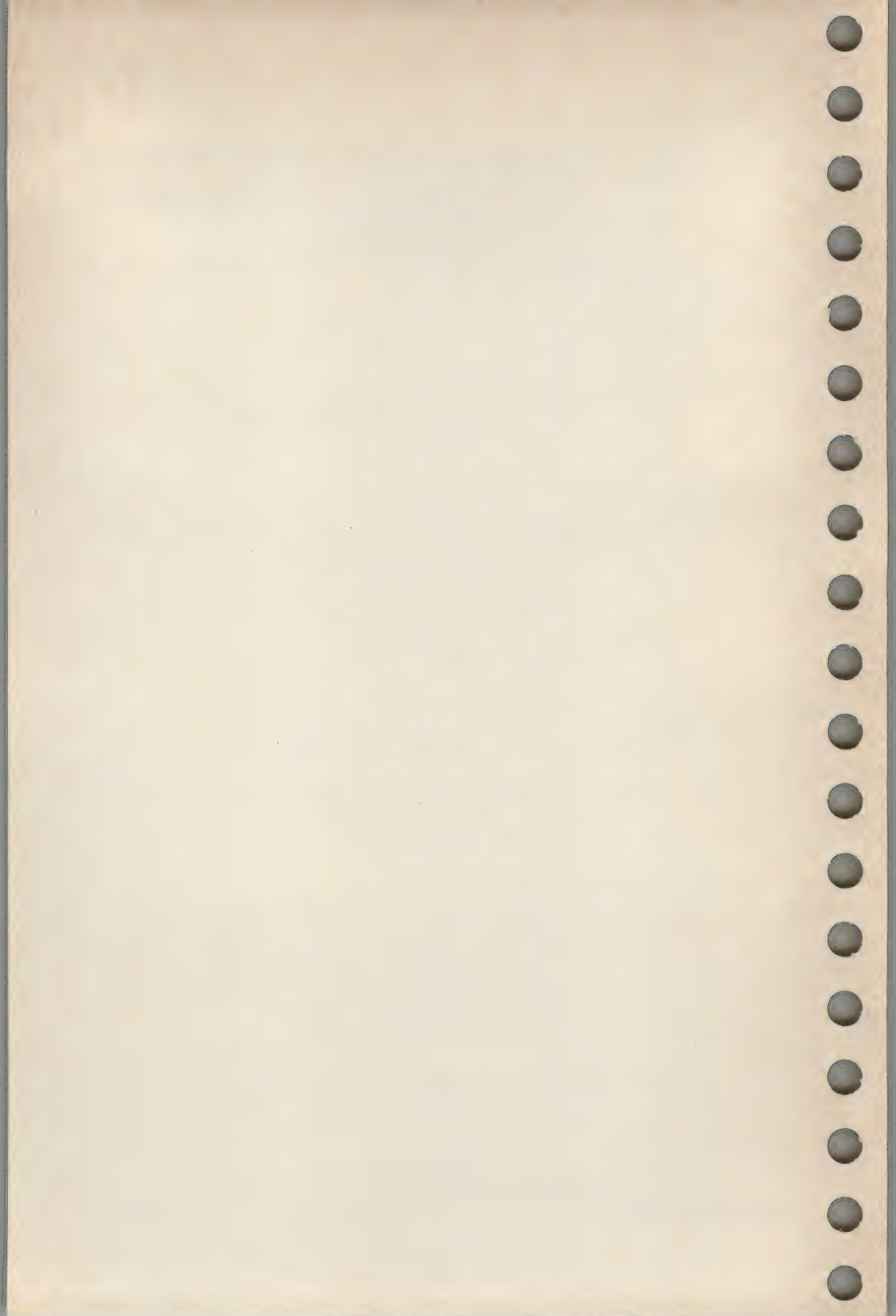
First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

An incorrect reference to the linker option `—a` has been corrected to `—e`.





## NAME

`cd` — change working directory

## SYNOPSIS

`cd [directory]`

## DESCRIPTION

If *directory* is not specified, the value of the environment variable HOME is used as the new working directory. If *directory* specifies a complete path starting with `/`, `.`, or `..`, *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the CDPATH environment variable. CDPATH has the same syntax as, and similar semantics to, the PATH variable, see *sh*(1). The command *cd* must have execute (search) permission in *directory*.

## APPLICATION USAGE

This is always a shell built in command.

## SEE ALSO

*pwd*(1), *sh*(1), *chdir*(2).

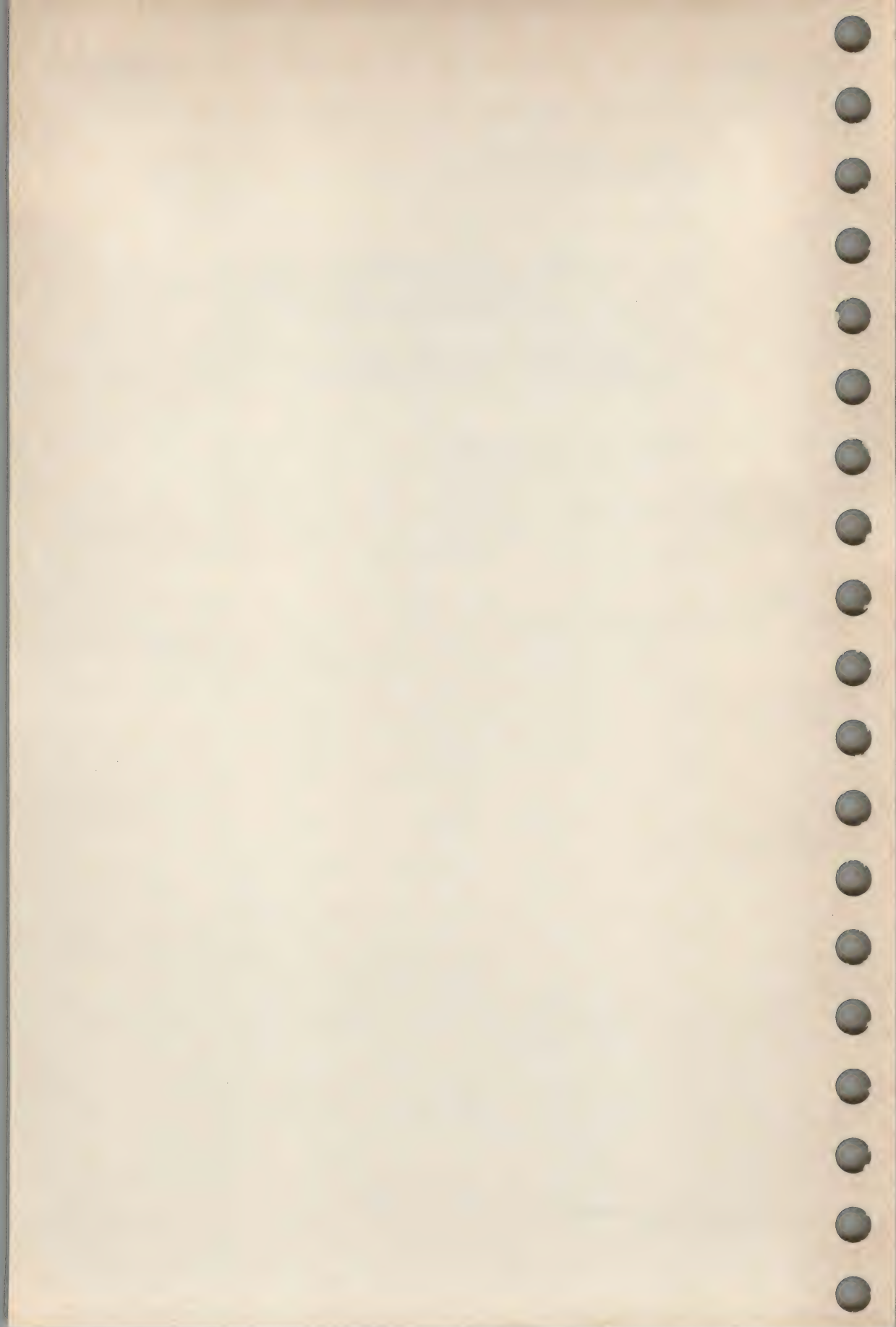
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`cflow` — generate C flowgraph

## SYNOPSIS

`cflow` [`—r`] [`—ix`] [`—i_`] [`—dnum`] *file*...

## DESCRIPTION

The command `cflow` analyses a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed in `.y`, `.l`, `.c`, and `.i` are yacc'd, lex'd, and C-preprocessed (bypassed for `.i` files) as appropriate, and then run through the first pass of `lint`. (The `—l`, `—D`, and `—U` options of the C-preprocessor are also understood by `cflow`.) Files suffixed with `.s` are assembled and information is extracted (as in `.o` files) from the symbol table. The output of all this processing is collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference (i.e., line) number, followed by a suitable amount of indentation indicating the level. This is followed by the name of the global (normally only a function not defined as an external or beginning with an underscore; see below for the `—i` inclusion option) a colon and its definition. For information extracted from C source, the definition consists of an abstract type declaration (e.g., `char*`) and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., `text`).

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only `<>` is printed.

The following options are interpreted by `cflow`:

- `—r` Reverse the caller:callee relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.
- `—ix` Include external and static data symbols. The default is to include only functions in the flowgraph.
- `—i_` Include names that begin with an underscore. The default is to exclude these functions (and data if `—ix` is used).
- `—dnum`

The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this is a very large number (typically greater than 32000). Attempts to set the cutoff depth to a non-positive integer will be ignored.



## EXAMPLE

Given the following in *file.c*:

```
int i;

main()
{
    f();
    g();
    f();
}

f()
{
    i = h();
}
```

The command

```
cflow -ix file.c
```

produces the output

```
1      main: int(), <file.c 4>
2      f: int(), <file.c 11>
3          h: <>
4          i: int, <file.c 1>
5      g: <>
```

## SEE ALSO

cc(1D), lex(1D), lint(1D), yacc(1D).

## APPLICATION USAGE

Files produced by *lex* and *yacc* cause the reordering of line number declarations which can confuse *cflow*. To get proper results, the input of *yacc* or *lex* must be directed to *cflow*.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The words "(typically greater than 32000)" have been added to the description of the *-d* option.

## NAME

chmod — change mode

## SYNOPSIS

chmod mode file...

## DESCRIPTION

The permissions of *file* or files are changed according to *mode*, which may be absolute or symbolic.

An absolute *mode* is a four-octal-digit number constructed from the logical *or* (sum) of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	Reserved
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0040	read by group
0020	write by group
0010	execute (search) by group
0004	read by others
0002	write by others
0001	execute (search) by others

A symbolic *mode* has the form:

[ who ] op permission [ op permission ]

The *who* part is a combination of the letters *u* (user), *g* (group) and *o* (other). The letter *a* stands for *ugo*, the default if *who* is omitted.

The argument *op* can be *+* to add *permission* to the file's mode, *-* to take away *permission* or *=* to assign *permission* absolutely (all other bits will be reset).

The argument *permission* is any combination of the letters *r* (read), *w* (write), *x* (execute) and *s* (set user or group ID); *u*, *g* or *o* indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with *=* to take away all permissions.

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter *s* is only useful with *u* or *g*.

Only the owner of a file (or the super-user) may change its mode. In order to set set-group-ID, the group of the file must correspond to the user's current group ID.



## EXAMPLES

1. This example denies write permission to others:

```
chmod o-w file
```

2. This example makes a file executable:

```
chmod +x file
```

## SEE ALSO

ls(1), umask(1), chmod(2).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The FUTURE DIRECTION on mandatory locking has been omitted.

## NAME

chown, chgrp — change owner or group

## SYNOPSIS

chown owner file...

chgrp group file...

## DESCRIPTION

## chown

The command *chown* changes the owner of *file* to *owner*. The owner may be either a decimal user ID or a login name found in the password file.

## chgrp

The command *chgrp* changes the group ID of *file* to *group*. The group may be either a decimal group ID or a group name found in the group file.

If either command is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode will be cleared.

## FILES

/etc/passwd	system's password file
/etc/group	system's group file

## SEE ALSO

chmod(1), chown(2).

## APPLICATION USAGE

Only the owner of a file or the super-user may change the owner or group of the file.

Some systems restrict the use of *chown* to the super-user.

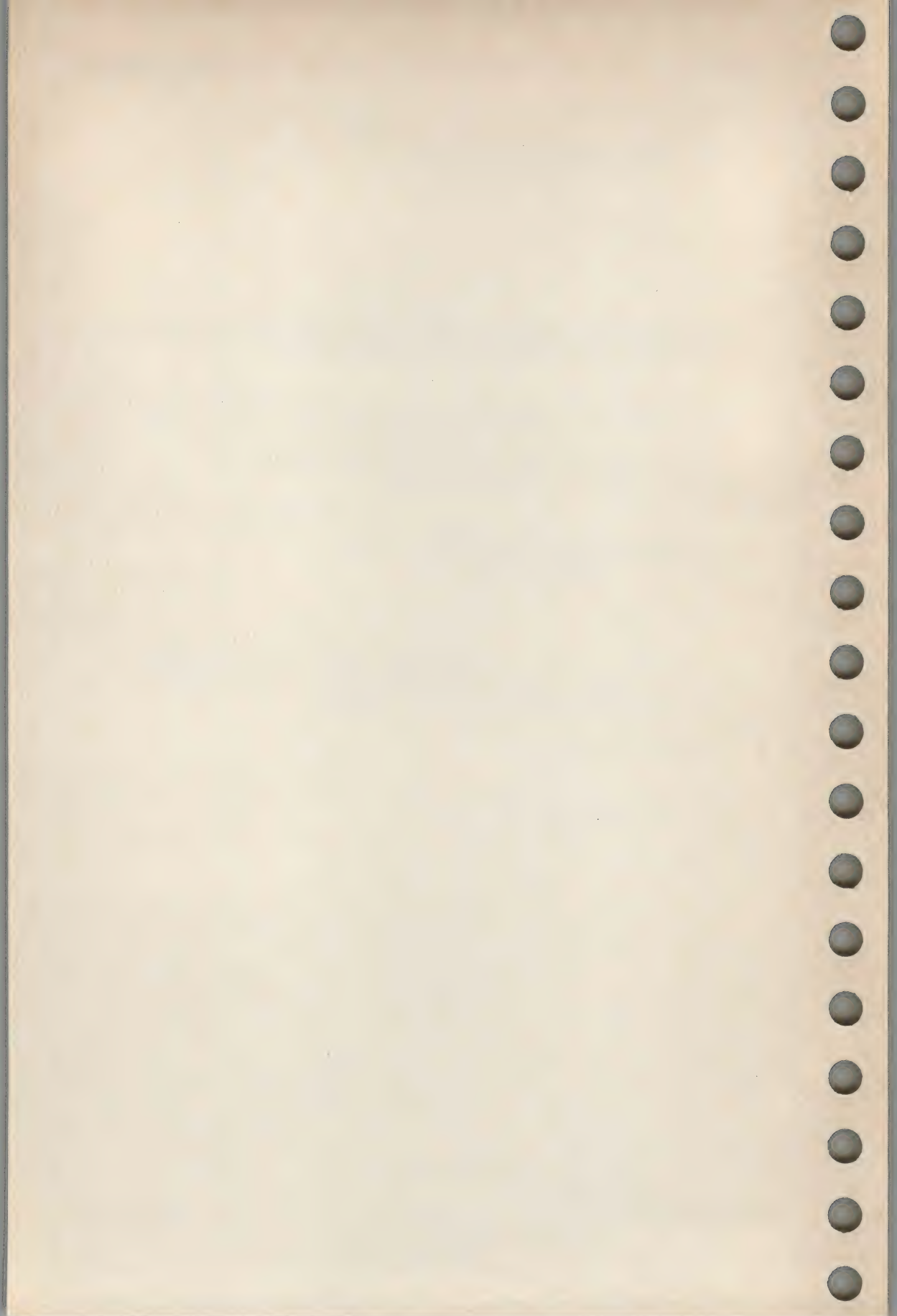
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





**NAME**

chroot — change root directory for a command

**SYNOPSIS**

chroot newroot command

**DESCRIPTION**

The command *chroot* executes the given *command*, with its root directory set to *newroot*. The meaning of any initial slashes (/) in path names is changed for a command and any of its children to *newroot*. Furthermore, the initial working directory is *newroot*.

This command is usually restricted to the super-user.

Notice that:

chroot newroot command >x

will create the file *x* relative to the original root, not the new one.

The new root path name is always relative to the current root: even if a *chroot* is currently in effect, the *newroot* argument is relative to the current root of the running process.

**SEE ALSO**

chdir(2).

**APPLICATION USAGE**

The user should note that many applications reference additional files which will need to be present in the new file tree.

**CHANGE HISTORY**

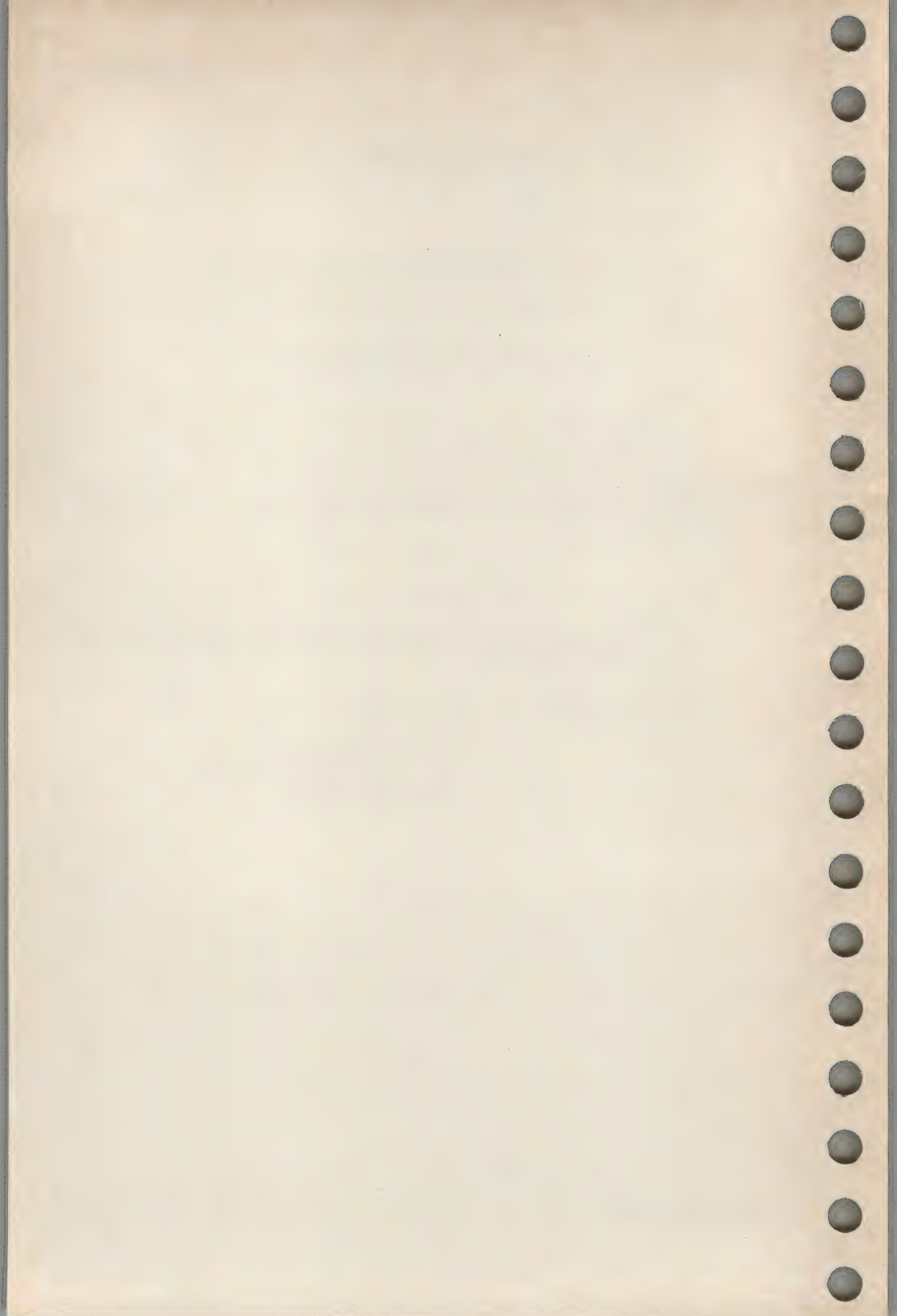
First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID with the following change:

The pathname */etc* has been removed from the SYNOPSIS.





## NAME

cmp — compare two files

## SYNOPSIS

cmp [-l] [-s] file1 file2

## DESCRIPTION

The command *cmp* compares two files. If *file1* is —, the standard input is used.

OF

Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is identical to the first part of the other, then it is reported that end-of-file was reached in the shorter file (before any differences were found).

Options:

OF

—l Print the byte number (decimal) and the differing bytes (octal) for each difference

—s Print nothing for differing files; return codes only.

## EXIT STATUS

Exit status is:

- 0 identical files
- 1 different files
- 2 inaccessible file or missing argument

## SEE ALSO

comm(1), diff(1).

## APPLICATION USAGE

Byte numbering begins at 1, rather than at 0.

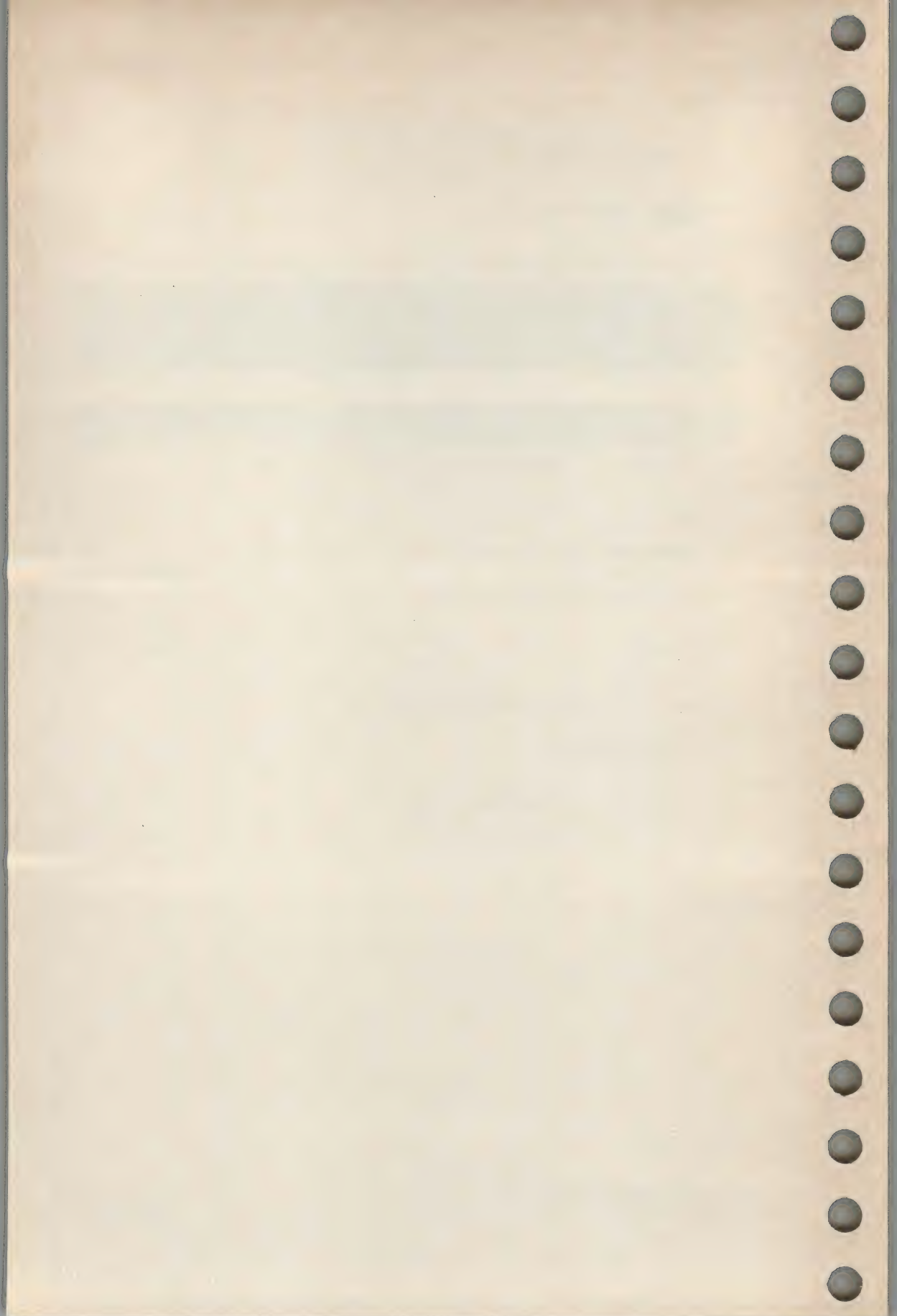
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

col — filter reverse line-feeds

## SYNOPSIS

col [ —bfpx ]

## DESCRIPTION

The command *col* reads from the standard input and writes to the standard output. It performs the line overlays implied by reverse line feeds, and by forward and reverse half-line feeds.

If the *—b* option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

Although *col* accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. This treatment can be suppressed by the *—f* (fine) option; in this case, the output from *col* may contain forward half-line feeds, but will still never contain either kind of reverse line motion.

Unless the *—x* option is given, *col* will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters SO and SI are assumed by *col* to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output SI and SO characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are <space>, <backspace>, <tab>, <return>, <newline>, SI, SO, VT, reverse line feed, forward half-line feed, and reverse half-line feed. The VT character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

The ASCII codes for the control functions and line-motion sequences mentioned above are as given in the table below. ESC stands for the ASCII *escape* character, with the octal code 033; ESC-*x* means a sequence of two characters, ESC followed by the character *x*.

reverse line feed	ESC-7
reverse half-line feed	ESC-8
forward half-line feed	ESC-9
vertical tab (VT)	013
start-of-text (SO)	016
end-of-text (SI)	017

Normally, *col* will remove any escape sequences found in its input that are unknown to it; the *—p* option may be used to force these to be passed through unchanged. The use of this option is discouraged unless the user is aware of the consequences.



**APPLICATION USAGE**

Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID.

## NAME

`comm` — select or reject lines common to two sorted files

## SYNOPSIS

```
comm [-[123]] file1 file2
```

OF

## DESCRIPTION

The command `comm` reads `file1` and `file2`, which should be ordered in the collating sequence of `sort`, see `sort(1)`, and produces a three-column output: lines only in `file1`; lines only in `file2`; and lines in both files. The file name `—` means the standard input.

Flags `1`, `2`, or `3` suppress printing of the corresponding column. Thus `comm —12` prints only the lines common to the two files; `comm —23` prints only lines in the first file but not in the second; `comm —123` is a no-op.

## SEE ALSO

`cmp(1)`, `diff(1)`, `sort(1)`, `uniq(1)`.

## CHANGE HISTORY

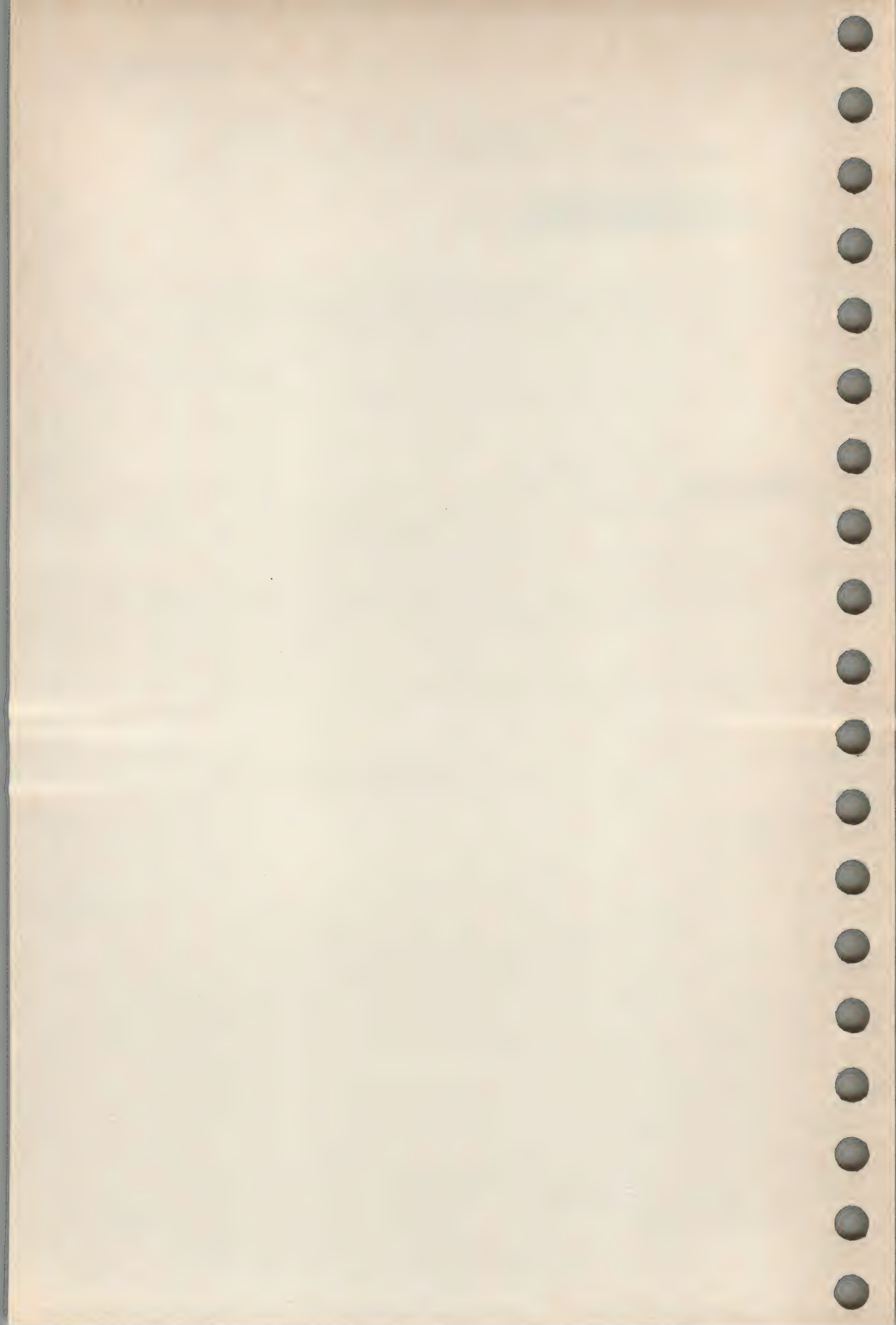
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

In the first sentence of the description, the words "ASCII collating sequence" have been changed to "the collating sequence of `sort`, see `sort(1)`".





## NAME

cp, ln, mv — copy, link or move files

## SYNOPSIS

cp file1 [file2...] target

ln [-f] file1 [file2...] target

mv [-f] file1 [file2...] target

## DESCRIPTION

These commands respectively copy, link, or move files; *file1* and *target* may not be the same. If *target* is not a directory, then only one file may be specified before it; if *target* is an existing file, its contents are destroyed, otherwise ( *target* is neither an existing file nor a directory) the file *target* is created. If *target* is a directory, then more than one file may be specified before it; the specified files are respectively copied, linked, or moved to that directory.

## cp

If *target* is not a directory, *cp* copies *file1* to *target*. If *target* exists, its contents are overwritten, but the mode, owner, and group are not changed. If *target* is a link to a file, all links remain (the file is changed).

If *target* is a directory, then the specified files are copied to that directory. For each *file* a new file, with the same mode, is created in the target directory; the owner and the group are those of the user making the copy.

## ln

If *target* is not a directory, *ln* links *file1* to *target*; that is, the name *target* is linked to the file *file1*. If *target* exists and its mode forbids writing, the mode is printed and the user asked for a response; if the response begins with a *y* (and the user is permitted) then the *ln* occurs. No questions are asked and the *ln* is done where permitted when the *-f* option is used or if the standard input is not a terminal.

If *target* is a directory, then the specified files are linked to that directory. That is, files with the same names are created in the directory, linked to the specified files.

## mv

If *target* is not a directory, *mv* moves (renames) *file1* as directed. If *target* does not exist, and has the same parent as *file1*, *file1* may be a directory: this allows a directory rename.

If *target* is a directory, then the specified files are moved to that directory.

If *file1* is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file.

If *target* is a file and its mode forbids writing, the mode is printed and the user asked for a response; if the response begins with a *y* (and the user is permitted) then the *mv* occurs. No questions are asked and the *mv* is done where permitted when the *-f* option is used or if the standard input is not a terminal.

## SEE ALSO

chmod(1), cpio(1), find(1), rm(1).



## APPLICATION USAGE

If *file1* and *target* lie on different file systems, *mv* may achieve the move by copying the file and deleting the original. In this case any linking relationship with other files is lost.

*Ln* will not link across file systems.

*Mv* can only be used to rename directories to a new name in the same parent. To rename directories to new parents, or to copy directories, use should be made of the *find*(1) and *cpio*(1) commands.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

`cpio` — copy file archives in and out

## SYNOPSIS

```
cpio —o [acBv]  
cpio —i [Bcdmrtuvf] [pattern...]  
cpio —p [adlmruv] directory
```

## DESCRIPTION

`cpio —o`

The command `cpio —o` (copy out) reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information. Output is padded to a 512-byte boundary.

`cpio —i`

The command `cpio —i` (copy in) extracts files from the standard input, which is assumed to be the product of a previous `cpio —o`. Only files with names that match *patterns* are selected. The arguments *patterns* are expressions making use of shell metanotation, with the exception of `[!...]`. In *patterns*, the metacharacters also match the `/` character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is `*` (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below. The permissions of the files will be those of the previous `cpio —o`. The owner and group of the files will be that of the current user unless the user is super-user, which causes `cpio` to retain the owner and group of the files of the previous `cpio —o`.

`cpio —p`

The command `cpio —p` (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination *directory* tree based upon the options described below.

The meanings of the available options are:

- a    Reset access times of input files after they have been copied. (When option `l` (see below) is also specified, the access times of the linked files are not reset.)
- B    Input/output is to be blocked 5120 bytes to the record (does not apply to the `—p` option; meaningful only with data directed to or from character special files).
- d    Directories are to be created as needed.
- c    Write or read header information in ASCII character form for portability.
- r    Interactively rename files. If the user types a null line, the file is skipped.
- t    Print a table of contents of the input. No files are created.
- u    Copy unconditionally (normally, an older file will not replace a newer file with the same name).
- v    Verbose: causes the names of the affected files to be printed. With the `t` option, provides a detailed listing.



- l** Whenever possible, link files rather than copying them. Usable only with the `—p` option.
- m** Retain previous file modification time. This option is ineffective on directories that are being copied.
- f** Copy in all files except those in *patterns*.

## EXAMPLES

1. Copy the contents of a directory onto a tape archive:

```
ls | cpio —oc >/dev/sctmtm0
```

2. Duplicate a directory hierarchy:

```
cd olddir  
find . —depth —print | cpio —pdl newdir
```

## SEE ALSO

ar(1), find(1), ls(1), tar(1).

## APPLICATION USAGE

Only the super-user can copy special files.

Archives created by *cpio* are portable between X/OPEN systems provided the same procedures are used. See "SOURCE CODE TRANSFER" for details.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The text "with the exception of *[!...]*" has been added in the DESCRIPTION.

## NAME

cpp — the C language preprocessor

## SYNOPSIS

cpp [ options ] [ ifile ] [ ofile ]

UN

## DESCRIPTION

The command *cpp* is the C language preprocessor, which is invoked as the first pass of any C compilation using the *cc* command. Thus the output of *cpp* is designed to be in a form acceptable as input to the next pass of the C compiler.

The *cpp* command optionally accepts two file names as arguments; *ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following options to *cpp* are recognised:

- P Preprocess the input without producing the line control information used by the next pass of the C compiler.
- C By default, *cpp* strips C-style comments. If the —C option is specified, all comments (except those found on *cpp* directive lines) are passed along.
- U*name*  
Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor.
- D*name*
- D*name*=*def*  
Define *name* as if by a *#define* directive. If no =*def* is given, *name* is defined as 1. The —D option has lower precedence than the —U option. That is, if the same name is used in both a —U option and a —D option, the name will be undefined regardless of the order of the options.
- I*dir* Change the algorithm for searching for *#include* files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, *#include* files whose names are enclosed in " " will be searched for first in the directory of the file with the *#include* line, then in directories named in —I options, and last in directories on a standard list. For *#include* files whose names are enclosed in <>, the directory of the file with the *#include* line is not searched.

Two special names are understood by *cpp*. The name `__LINE__` is defined as the current line number (as a decimal integer) as known by *cpp*, and `__FILE__` is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by *#*. Any number of blanks and tabs are allowed between the *#* and the directive. The directives are:

**#define** *name token-string*

Replace subsequent instances of *name* with *token-string*.



**#define** *name*( *arg*, ..., *arg* ) *token-string*

Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a "(", a list of comma-separated set of tokens, and a ")" by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, *cpp* re-starts its scan for names to expand at the beginning of the newly created *token-string*.

**#undef** *name*

Cause the definition of *name* (if any) to be forgotten from now on. No additional tokens are permitted on the line after *name*.

**#include** " *filename* "**#include** <*filename*>

Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the *-I* option above for more detail. No additional tokens are permitted on the line after the final " or >.

**#line** *integer-constant* " *filename* "

Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. If *filename* is not given, the current file name is unchanged. No additional tokens are permitted on the line after the final ".

**#endif**

Ends a section of lines begun by a test directive (*#if*, *#ifdef* or *#ifndef*). Each test directive must have a matching *#endif*. No additional tokens are permitted on the line.

**#ifdef** *name*

The lines following will appear in the output if and only if *name* has been the subject of a previous *#define* without being the subject of an intervening *#undef*. No additional tokens are permitted on the line after *name*.

**#ifndef** *name*

The lines following will not appear in the output if and only if *name* has been the subject of a previous *#define* without being the subject of an intervening *#undef*. No additional tokens are permitted on the line after *name*.

**#if constant-expression**

Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the `?:` operator, the unary `—`, `!`, and `~` operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator *defined*, which can be used in *constant-expression* in these two forms: *defined(name)* or *defined name*. This allows the utility of *#ifdef* and *#ifndef* in a *#if* directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the *sizeof* operator is not available.

**#else**

The else part of an *#ifdef*, *#ifndef*, or *#if*. The lines preceding are ignored, and the lines following (up to the *#endif*) are included in the output if the test is false.

The test directives and the optional *#else* directives can be nested.

**EXAMPLE**

In order to test whether either of the two symbols *abc* and *def* are defined, use

```
#if defined (abc) || defined (def)
```

**SEE ALSO**

*cc*(1D), *m4*(1D).

**APPLICATION USAGE**

The recommended way to invoke *cpp* is through the *cc* command. See *m4*(1D) for a general macro processor.

Include directives should avoid using hard-coded path-names: for example,

```
#include <file.h>
```

should be used, rather than

```
#include "/usr/include/file.h"
```

**FUTURE DIRECTIONS**

The SVID reserves the `—Y` option for future use. It will be used to specify a directory to be used instead of the standard list, when searching for *#include* files.

Users will also be able to specify, by means of the *TMPDIR* environment variable, the directory in which any temporary files are to be created.

**CHANGE HISTORY**

First released in Issue 2.

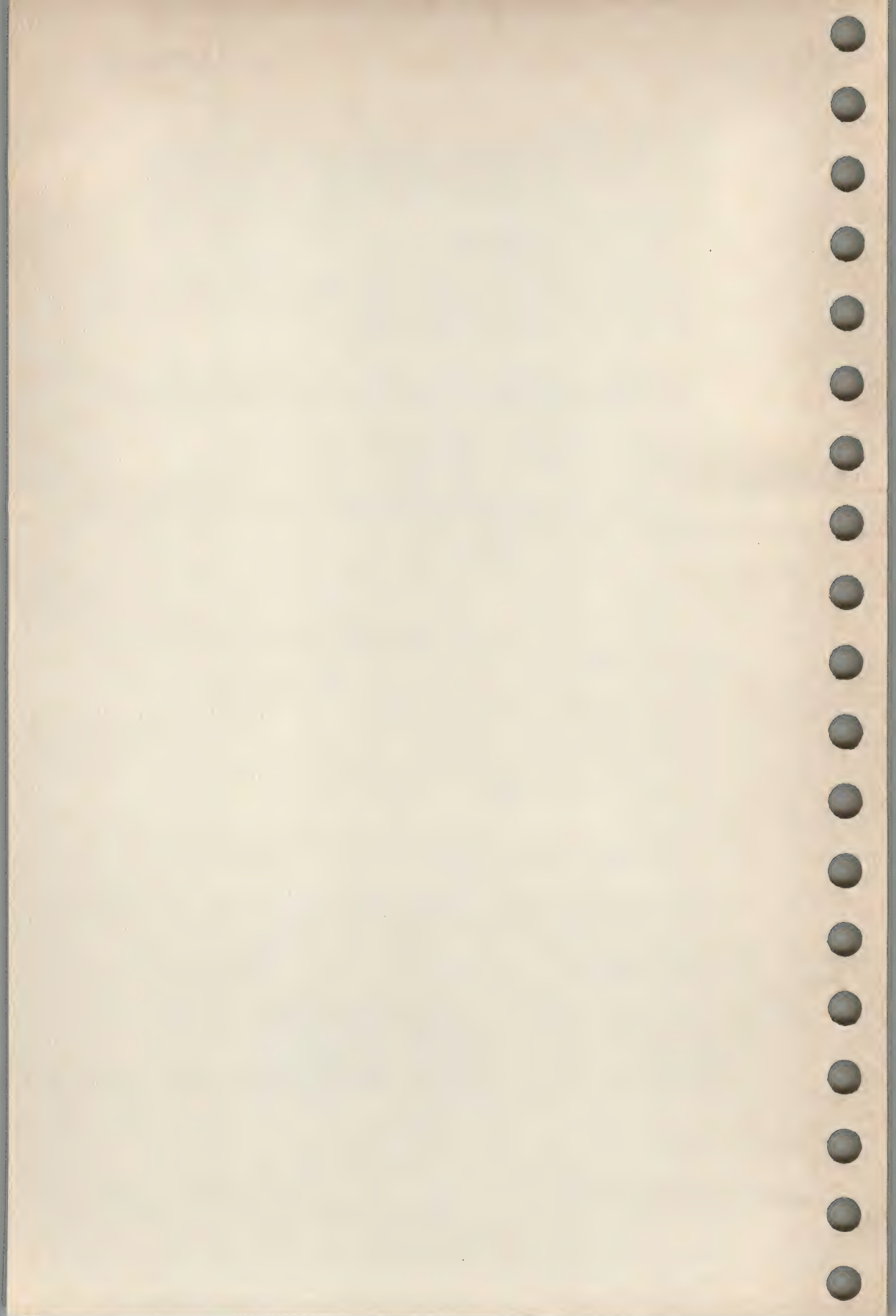
**Issue 2**

Derived from the entry in Issue 2 of the SVID with the following changes:

The pathname *LIBDIR* has been removed from the *SYNOPSIS* line.

An incorrect `@` operator in the definition of *#if constant-expression* has been changed to `~`.





## NAME

crontab — user crontab file

## SYNOPSIS

crontab [ file ]

crontab -r

crontab -l

## DESCRIPTION

The command *crontab* copies the specified file, or standard input if no file is specified, into a directory that holds all users' *crontab* files. The *-r* option removes a user's *crontab* file from the crontab directory. The option *-l* will list the *crontab* file of the invoking user.

Users are permitted to use *crontab* if their names appear in the file */usr/lib/cron/cron.allow*. If that file does not exist, the file */usr/lib/cron/cron.deny* is checked to determine if the user should be denied access to *crontab*. If neither file exists, only the super-user is allowed to submit a job. If only *cron.deny* exists and is empty, global usage is permitted. The allow/deny files consist of one user name per line.

A *crontab* file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0-59),  
hour (0-23),  
day of the month (1-31),  
month of the year (1-12),  
day of the week (0-6 with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, each one is effective independent of the other. For example, *0 0 1,15 \* 1* would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to *\** (for example, *0 0 \* \* 1* would run a command only on Mondays).

The sixth field of a line in a *crontab* file is a string that is executed by the command interpreter at the specified times. A percent character in this field (unless escaped by *\*) is translated to a newline character. Only the first line (up to a % or end of line) of the command field is executed by the command interpreter. The other lines are made available to the command as standard input. The event scheduler supplies a default environment, defining the environment variables HOME, LOGNAME, and PATH.

**Note:** If standard output and standard error are not redirected, any generated output or errors will be mailed to the user.



# CRONTAB(1)

Utilities

## FILES

/usr/lib/cron/cron.allow	list of allowed users
/usr/lib/cron/cron.deny	list of denied users

## SEE ALSO

sh(1).

## APPLICATION USAGE

The new *crontab* file for a user overwrites an existing one.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

csplit — context split

## SYNOPSIS

csplit [**—s**] [**—k**] [**—f**prefix] file arg1 [arg2...argn]

## DESCRIPTION

The command *csplit* reads *file* and separates it into  $n+1$  sections, defined by the arguments *arg1*...*argn*. By default the sections are written to *xx00*...*xxn* ( $n$  may not be greater than 99). These sections get the following pieces of *file*:

- 00: From the start of *file* up to (but not including) the line referenced by *arg1*.
- 01: From the line referenced by *arg1* up to the line referenced by *arg2*.

$n$ : From the line referenced by *argn* to the end of *file*.

If the *file* argument is a **—** then standard input is used.

The options to *csplit* are:

- s** *Csplit* normally writes the character counts for each file created. If the **—s** option is present, *csplit* suppresses the printing of all character counts.
- k** *csplit* normally removes created files if an error occurs. If the **—k** option is present, *csplit* leaves previously created files intact.
- f**prefix  
If the **—f** option is used, the created files are named *prefix00* ... *prefixn*. The default is *xx00* ... *xxn*.

The arguments (*arg1* ... *argn*) to *csplit* can be a combination of the following:

**/re/** A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *re*. Simple regular expression syntax is accepted. The current line becomes the line containing *re*. This argument may be followed by an optional **+** or **-** some number of lines (e.g., **/Page/-5**).

**%re%**

This argument is the same as **/re/**, except that no file is created for the section.

**line\_no**

A file is to be created from the current line up to (but not including) the line number *line\_no*. The current line becomes *line\_no*.

**{num}** Repeat argument. This argument may follow any of the above arguments. If it follows a *re* type argument, that argument is applied *num* more times. If it follows *line\_no*, the file will be split every *line\_no* lines (*num* times) from that point.



All *re* type arguments that contain blanks or other characters meaningful to the command interpreter must be enclosed in the appropriate quotes. Regular expressions may not contain embedded newlines. The command *csplit* does not affect the original file; it is the user's responsibility to remove it.

## EXAMPLES

1. This example creates four files, *cobol00* ... *cobol03*:

```
csplit —fcobol file '/PROCEDURE DIVISION/' /par5./ /par16./
```

After editing the split files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that this example overwrites the original file.

2. This example splits the file at every 100 lines, up to 10000 lines:

```
csplit —k file 100 {99}
```

The `—k` option causes the created files to be retained if there are less than 10000 lines; however, an error message would still be printed.

3. Assuming that *prog.c* follows the normal C coding convention of ending routines with a `}` at the beginning of the line, this example will create a file containing each separate C routine (up to 21) in *prog.c*.

```
csplit —k prog.c '%main(%' '/'^)/+1' {20}
```

## ERRORS

An error is reported if an argument does not reference a line between the current position and the end of the file.

## SEE ALSO

*sh*(1).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

An erroneous reference to *n* + 1 in the first paragraph has been corrected.

An erroneous file name "*cobol08*" has been corrected to "*cobol03*" in the first example.

## NAME

*cu* — call another system

## SYNOPSIS

```
cu [—s speed] [—l line] [—h] [—t] [—d] [—o | —e] [—n] telno
cu [—s speed] [—h] [—d] [—o | —e] —l line
cu [—h] [—d] [—o | —e] systemname
```

UN

UN

UN

## DESCRIPTION

The command *cu* calls up another system. It manages an interactive conversation, with possible transfers of ASCII files.

The command *cu* accepts the following options and arguments:

*—s speed*

Specifies the transmission speed. The default value is "Any" speed which will depend on the order of the lines in the system devices file.

*—l line*

Specifies a device name to use as the communication line. This can be used to override the search that would otherwise take place for the first available line having the right speed. When the *—l* option is used without the *—s* option, the speed of a line is taken from the devices file. When the *—l* and *—s* options are both used together, *cu* will search the devices file to check if the requested speed for the requested line is available. If so, the connection will be made at the requested speed; otherwise, an error message will be printed and the call will not be made. If the specified device is associated with an auto dialler, a telephone number must be provided. Use of this option with *systemname* rather than *telno* is not allowed (see *systemname* below).

*—h* Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

*—t* Used to dial an ASCII terminal which has been set to auto answer. Appropriate mapping of <carriage-return> to <carriage-return-line-feed> pairs is set.

*—d* Causes diagnostic traces to be printed.

*—o* Designates that odd parity is to be generated for data sent to the remote system.

*—e* Designates that even parity is to be generated for data sent to the remote system.

*—n* For added security, prompt the user to provide the telephone number to be dialled rather than taking it from the command line.

*telno* When using an automatic dialler, the argument is the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of 4 seconds.

*systemname*

A *uucp* system name may be used rather than a telephone number; in this case, *cu* will obtain an appropriate direct line or telephone number from a system file.



**Note:** the *systemname* option should not be used in conjunction with the *—l* and *—s* options as *cu* will connect to the first available line for the system name specified ignoring the requested line and speed.

After making the connection, *cu* runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with *~*, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with *~*, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with *~* have special meanings.

The *transmit* process interprets the following user initiated commands:

- ~* terminate the conversation.
- ~!* escape to an interactive command interpreter on the local system.
- ~!cmd...*  
execute *cmd* on the local system.
- ~\$cmd...*  
run *cmd* locally and send its output to the remote system for execution.
- ~%cd* change the directory on the local system.
- ~%take from [to]*  
copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
- ~%put from [to]*  
copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places.
- ~~line* send the line *~line* to the remote system.
- ~%break*  
transmit a BREAK to the remote system (which can also be specified as *~%b*).
- ~%nostop*  
toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output.

The use of *~%put* requires *stty(1)* and *cat(1)* on the remote side. It also requires that the current *<erase>* and *<kill>* characters on the remote system be identical to these current control characters on the local system. Backslashes are inserted at appropriate places.

The use of *~%take* requires the existence of *echo(1)* and *cat* on the remote system. Also, *tabs* mode (see *stty(1)*) should be set on the remote system if tabs are to be copied without expansion to spaces.

When *cu* is used on system *X* to connect to system *Y* and subsequently used on system *Y* to connect to system *Z*, commands on system *Y* can be executed by using *^^*. For example, *uname* can be executed on *Z*, *X*, *Y* as follows (the response is given in italics):

```
uname
Z
^[X]!uname
X
^^[Y]!uname
Y
```

In general, *~* causes the command to be executed on the original machine; *^^* causes the command to be executed on the next machine in the chain.

#### EXAMPLES

1. To dial a system whose telephone number is 9 1 201 555 1212 using 1200 baud (where dial tone is expected after the 9):

```
cu -s 1200 9=12015551212
```

If the speed is not specified, "Any" is the default value.

2. To log in to a system connected by a direct line:

```
cu -l /dev/ttyXX
```

or

```
cu -l ttyXX
```

3. To dial a system with the specific line and a specific speed:

```
cu -s 1200 -l ttyXX
```

4. To dial a system using a specific line associated with an auto dialler:

```
cu -l cu1XX 9=12015551212
```

5. To use a system name:

```
cu systemname
```

#### SEE ALSO

*cat*(1), *echo*(1), *stty*(1), *uname*(1), *uucp*(1).

#### APPLICATION USAGE

Typical implementations of this utility require a communications line configured to use the *termio*(7) interface. On systems where none of these lines are available, this utility may not be present.



## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The words "calls up another system, which will usually be a System V system, but may be a terminal, or a non-System V system." have been changed to "calls up another system."

In the SVID, brackets were used to highlight responses in the *uname* example. To avoid confusion with brackets which are part of the response, the relevant parts of that example are shown here in *italics*.

## NAME

`cut` — cut out selected fields of each line of a file

## SYNOPSIS

`cut —clist [ file... ]`

`cut —flist [ —dchar ] [ —s ] [ file... ]`

## DESCRIPTION

The command *cut* cuts out columns from a table or fields from each line of a file. The fields as specified by *list* can be of fixed length, specified by character position (`—c` option), or the length can vary from line to line and be marked with a field delimiter character like tab (`—f` option). The command *cut* can be used as a filter; if no *files* are given, the standard input is used.

The option qualifier *list* (see options `—c` and `—f` below) is a comma-separated list of integers (in increasing order), with optional `-` to indicate ranges; e.g., `1,4,7`; `1-3,8`; `-5,10` (short for `1-5,10`); or `3-` (short for third through last).

The meanings of the options are:

`—clist`

The *list* following `—c` (no space) specifies character positions (e.g., `—c1-72` would pass the first 72 characters of each line).

`—flist`

The *list* following `—f` is a list of fields assumed to be separated in the file by a delimiter character (see `—d`); e.g., `—f1,7` copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings) unless `—s` is specified.

`—dchar`

The character following `—d` is the field delimiter (used with the `—f` option only). Default is the tab character. Space or other characters with special meaning to the command interpreter must be quoted.

`—s` Suppresses lines with no delimiter characters when used with the `—f` option. Unless specified, lines with no delimiters will be passed through untouched.

Either the `—c` or the `—f` option must be specified.

## EXAMPLES

The following maps user IDs to names:

```
cut —d: —f1,5 /etc/passwd
```

## SEE ALSO

`grep(1)`, `paste(1)`, `sh(1)`.

## APPLICATION USAGE

*Grep* can be used to make horizontal “cuts” (by context) through a file, and *paste* to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste* together.



# CUT(1)

*Utilities*

## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

`cxref` — generate C program cross-reference

## SYNOPSIS

`cxref [ options ] file ...`

## DESCRIPTION

The command `cxref` analyses a collection of C *files* and attempts to build a cross-reference table. Information from `#define` lines is included in the symbol table. A listing is produced on standard output of all symbols (auto, static, and global) in each *file* separately, or with the `—c` option, in combination. Each symbol contains an asterisk (\*) before the declaring reference.

In addition to the `—D`, `—I`, and `—U` options (which are identical to their interpretation by `cc`) the following options are interpreted by `cxref`:

`—c` Print a combined cross-reference of all input files.

`—wnum`

Width option which formats output no wider than *num* (decimal) columns. This option will default to 80 if *num* is not specified or is less than 51.

`—o file`

Direct output to named *file*.

`—s` Operate silently; does not print input file names.

## SEE ALSO

`cc(1D)`.

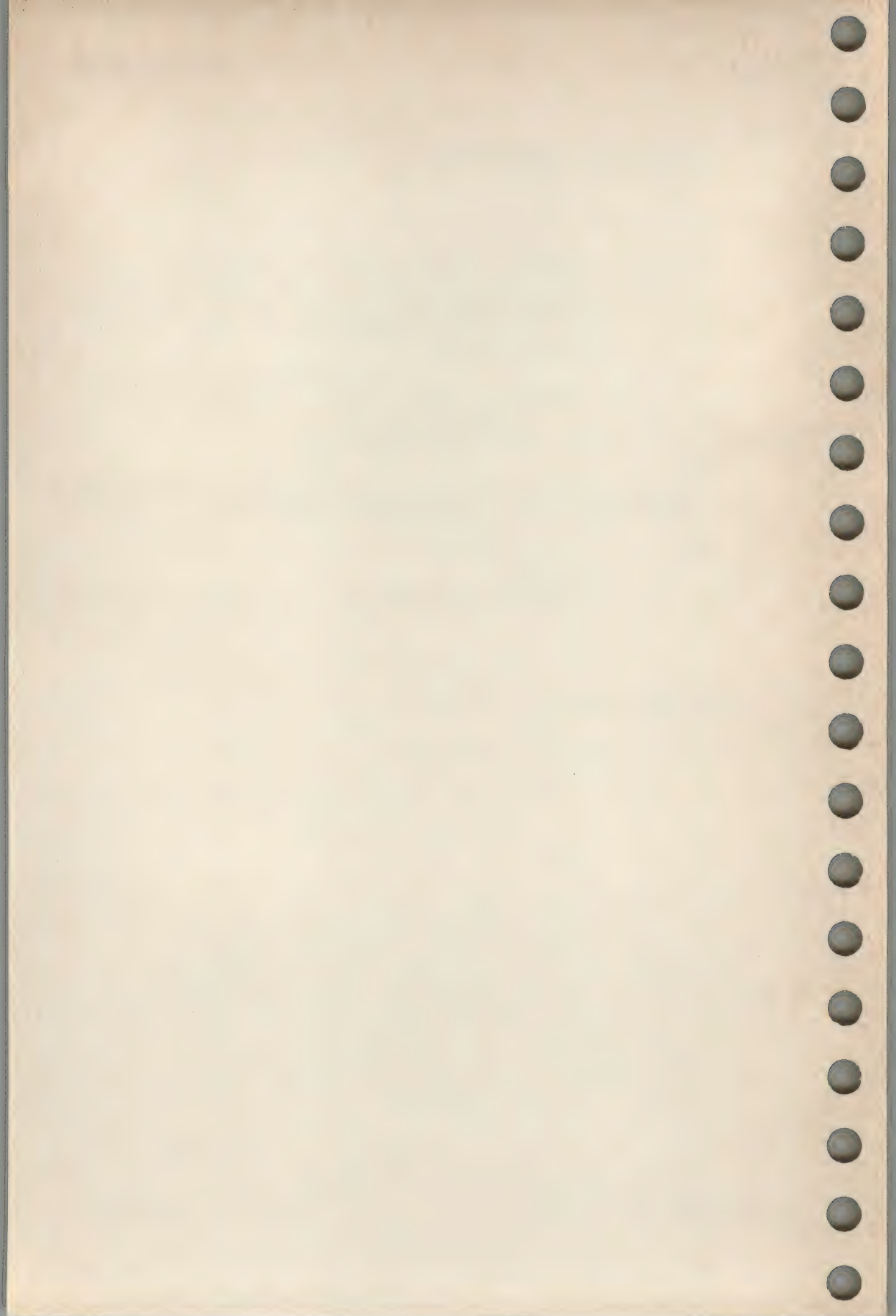
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

date — print the date

## SYNOPSIS

date mmddhhmm[yy]

UN

date [ +format ]

## DESCRIPTION

OF LA If no argument is given, the current date and time are printed.

UN If an argument in the form mmddhhmm[yy] is given, date attempts to set the system date from the value given in the argument. This is only possible if the user is super-user and the system permits the setting of the system date. The first mm is the month (number); dd is the day (number); hh is the hour (number, 24 hour system); the second mm is the minute (number) and yy is the last two digits of the year and is optional. For example:

date 10080045

sets the date to October 8, 00:45. The current year is the default if yy is omitted.

The system operates in GMT; date takes care of conversion to and from the local time. If present, the environmental variable TZ is used to override the default time zone, see ctime(3C).

If an argument beginning with + is given, the output of date is under the control of the user. The format for the output is similar to that of the first argument to the printf routine, see printf(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a newline character.

## Field Descriptors:

n insert a newline character  
 t insert a tab character  
 m month of year — 01 to 12  
 d day of month — 01 to 31  
 y last 2 digits of year — 00 to 99  
 D date as mm/dd/yy  
 H hour — 00 to 23  
 M minute — 00 to 59  
 S second — 00 to 59  
 T time as HH:MM:SS  
 j day of year — 001 to 366  
 w day of week — Sunday = 0

LA

a abbreviated weekday  
 h abbreviated month  
 i time in AM/PM notation



# DATE(1)

Utilities

## EXAMPLE

The following generates output as shown:

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

Output:

```
DATE: 08/01/76
```

```
TIME: 14:45:05
```

## SEE ALSO

ctime(3C), printf(3S).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

English Language descriptions of the abbreviated day and month names for the `—a` and `—h` options have been removed.

## NAME

dd — convert and copy a file

## SYNOPSIS

dd [[ option=value ] ...]

## DESCRIPTION

The command *dd* copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

Option	Values
<i>if=file</i>	input file name; standard input is default.
<i>of=file</i>	output file name; standard output is default.
<i>ibs=n</i>	input block size <i>n</i> bytes (default 512).
<i>obs=n</i>	output block size (default 512).
<i>bs=n</i>	set both input and output block size, superseding <i>ibs</i> and <i>obs</i> ; also, if no conversion is specified, it is particularly efficient since no in-core copy need be done.
<i>cbs=n</i>	conversion buffer size.
<i>skip=n</i>	skip <i>n</i> input blocks before starting copy.
<i>seek=n</i>	seek <i>n</i> blocks from beginning of output file before copying.
<i>count=n</i>	copy only <i>n</i> input blocks.
<i>conv=ascii</i>	convert EBCDIC to ASCII.
<i>ebcdic</i>	convert ASCII to EBCDIC.
<i>ibm</i>	slightly different map of ASCII to EBCDIC.
<i>lcase</i>	map alphabetics to lower case.
<i>ucase</i>	map alphabetics to upper case.
<i>swab</i>	swap every pair of bytes.
<i>noerror</i>	do not stop processing on an error.
<i>sync</i>	pad every input block to <i>ibs</i> .
...,...	several comma-separated conversions.

Where sizes are specified, a number of bytes is expected. A number may end with *k*, *b*, or *w* to specify multiplication by 1024, 512, or 2, respectively; a pair of numbers may be separated by *x* to indicate a product.

The option *cbs* is used only if ASCII or EBCDIC conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and <newline> added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output block of size *cbs*.

After completion, *dd* reports the number of whole and partial input and output blocks.



**EXAMPLE**

This command will read an EBCDIC tape blocked ten 80-byte EBCDIC card images per block into the ASCII file *x* :

```
dd if=/dev/rsctmtm0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw source code transfer device. The command *dd* is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary block sizes.

**APPLICATION USAGE**

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC.

More than one variant of the EBCDIC character set exists.

**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID.

## NAME

delta — make a delta (change) to an SCCS file

## SYNOPSIS

delta [—rSID] [—s] [—n] [—glist] [—m[mrlist]] [—y[comment]] [—p] file ...

## DESCRIPTION

The command *delta* is used to permanently introduce into the named SCCS *file* changes that were made to the file retrieved by *get* (called the *g-file*, or generated file).

The *delta* command makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with *s.*) and unreadable files are silently ignored. If a name of — is given, the standard input is read; in this case the —y option (see below) is required on the command line; if the —m option (see below) would normally be required, then it too is required on the command line. Each line of the standard input is taken to be the name of an SCCS file to be processed.

The *delta* command may issue prompts on the standard output depending upon certain options specified and flags, see *admin*(1D), that may be present in the SCCS *file* (see —m and —y options below).

Lines beginning with an SOH ASCII character (binary 001) cannot be placed in the SCCS file unless the SOH is escaped. This character has special meaning to SCCS and will cause an error.

Keyletter arguments apply independently to each named file.

—rSID Uniquely identifies which delta is to be made to the SCCS file. The use of this option is necessary only if two or more outstanding *gets* for editing (*get -e*) on the same SCCS file were done by the same person (login name). The SID value specified with the —r option can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command, see *get*(1D). A diagnostic results if the specified SID is ambiguous, or if it is necessary and omitted on the command line.

—s Suppresses the issue on the standard output of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

—n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

—glist

Specifies a *list*, see *get*(1D) for the definition of *list*, of deltas which are to be ignored when the file is accessed at the change level (SID) created by this delta.

—m[mrlist]

If the SCCS file has the v flag set, see *admin*(1D), then a Modification Request (MR) number must be supplied as the reason for creating the new delta.

If —m is not used and the standard input is a terminal, the prompt "MRs?" is issued on the standard output before the standard input is read; if the standard



input is not a terminal, no prompt is issued. The "MRs?" prompt always precedes the *comments?* prompt (see *—y* option).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the *v* flag has a value, it is taken to be the name of a program which will validate the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, *delta* terminates. (It is assumed that the MR numbers were not all valid.)

## *—y*[comment]

Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

If *—y* is not specified and the standard input is a terminal, the prompt "*comments?*" is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the comment text.

## *—p*

Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in *diff* format, see *diff*(1).

## SEE ALSO

*admin*(1D), *get*(1D), *prs*(1D), *rmdel*(1D).

## APPLICATION USAGE

Some systems limit the size of text to *—y*[comment]; if present, this limit will not be less than 512 bytes.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

`df` — report free disk space

## SYNOPSIS

`df` [`-t`] [*file-system*...]

OF

## DESCRIPTION

The command `df` prints out the free space (in 512-byte units) and the number of free file slots ("inodes") available, for on-line file systems. The argument *file-system* may be specified either by device name (e.g., `/dev/dsk/0s1`) or by mounted directory name (e.g., `/usr`). If no *file-system* is specified, the free space on all of the mounted file systems is printed.

UN

The `-t` option causes the total allocated space figures to be reported as well.

## APPLICATION USAGE

Currently not all systems report in terms of 512-byte units. This situation may change in the future.

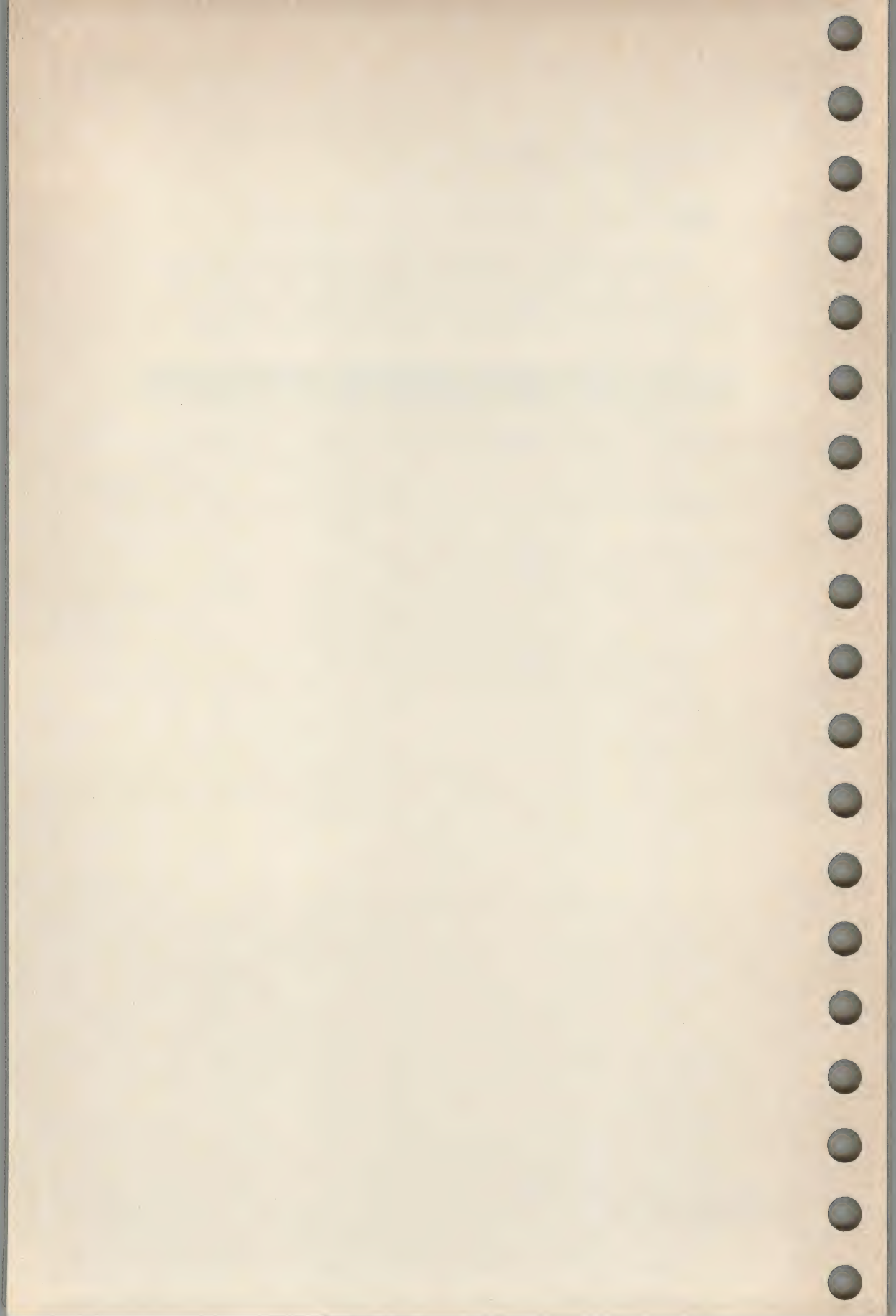
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

diff — differential file comparator

## SYNOPSIS

diff [—efb<sub>n</sub>] file1 file2

## DESCRIPTION

The command *diff* tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is —, the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory with the name *file2* (*file1*) is used. The normal output contains lines of these forms:

n1 a n3,n4

n1,n2 d n3

n1,n2 c n3,n4

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging *a* for *d* and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs, where *n1* = *n2* or *n3* = *n4*, are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by <, then all the lines that are affected in the second file flagged by >.

The —*b* option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The —*e* option produces a script of *a*, *c*, and *d* commands for the editor *ed*, which will recreate *file2* from *file1*.

The —*f* option produces a similar script, not useful with *ed*, in the opposite order.

PI

Option —*h* does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

PI

Options —*e* and —*f* are unavailable with the —*h* option.

## EXIT STATUS

Exit status is:

- 0 no differences
- 1 differences
- 2 errors

## SEE ALSO

cmp(1), comm(1), ed(1).

## APPLICATION USAGE

Editing scripts produced under the —*e* or —*f* option may be incorrect when dealing with lines consisting of a single period.



## DIFF(1)

*Utilities*

### CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

`dircmp` — directory comparison

## SYNOPSIS

`dircmp [-d] [-s] dir1 dir2`

OF

## DESCRIPTION

The command *dircmp* examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is specified, a list is output indicating whether the file names common to both directories have the same contents.

- d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).
- s Suppress messages about identical files.

## SEE ALSO

*cmp*(1), *diff*(1).

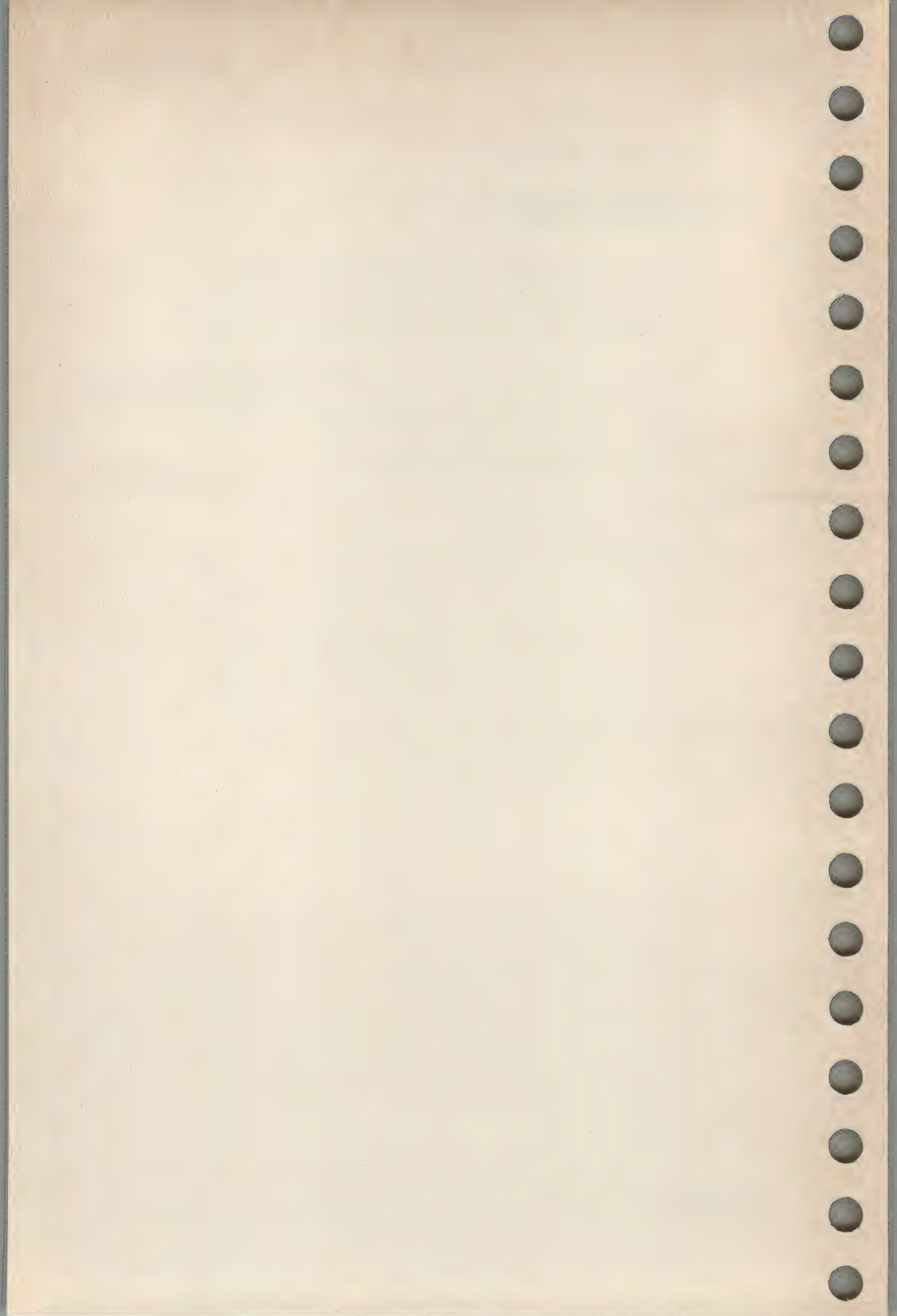
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

dis — disassembler (OPTIONAL)

## SYNOPSIS

dis [**—o**] [**—V**] [**—L**] [**—F** *function*] [**—I** *string*] *file* . .

PI

## DESCRIPTION

The *dis* command produces an assembly language listing of each of its *file* arguments, each of which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

The following options are interpreted by the disassembler and may be specified in any order.

**—o** Will print numbers in octal. Default is hexadecimal.

MV

**—V** Version number of the disassembler will be written to standard error.

**—L** Invokes a lookup of C source labels in the symbol table for subsequent printing.

**—F** *function*

Disassembles only the named *function* in each object file specified on the command line. This option may be specified a number of times on the command line.

**—I** *string*

Will disassemble the library file specified as *string*. For example, the command *dis -lm* will disassemble the math library.

## SEE ALSO

as(1D), cc(1D).

## FUTURE DIRECTIONS

The **—s** option is reserved for future use. It will be used to specify symbolic disassembly.

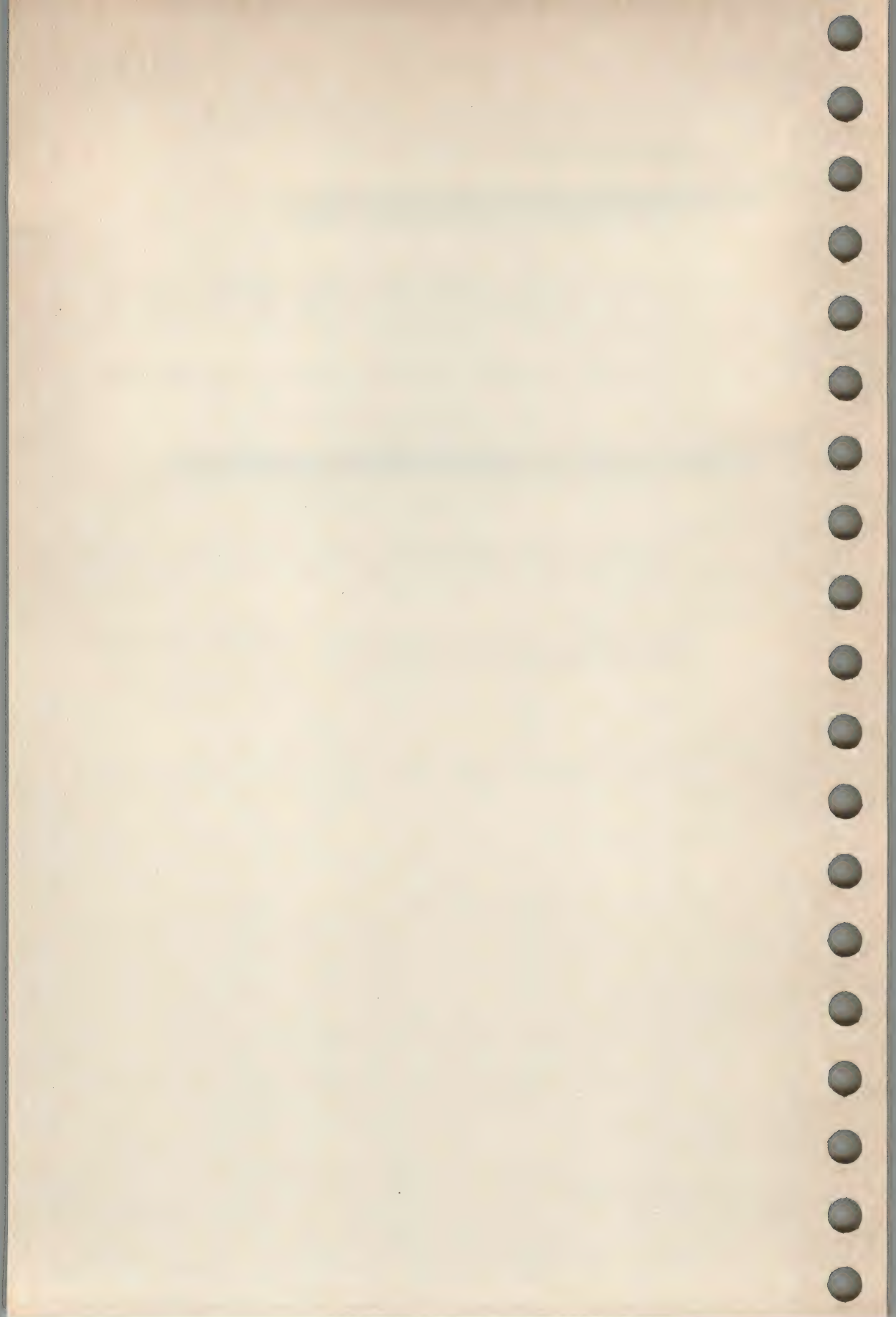
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`du` — estimate file space usage

## SYNOPSIS

`du [—ars] [file...]`

OF PI

## DESCRIPTION

The command `du` gives an estimate, in 512-byte units, of the file space contained in all the specified *files*. Whenever a directory is named, all files within it are reported; sub-directories are traversed recursively. If no *file* is specified, the current directory is used.

The option `—s` causes only the grand total (for each of the specified *files*) to be given. The option `—a` causes a report to be generated for each *file*. With no options, a report is given for each directory only.

## UN

Some implementations of `du` are silent about directories that cannot be read, files that cannot be opened, etc. If this is the case, the `—r` option will cause `du` to generate messages in such instances.

A file with two or more links is only counted once.

## APPLICATION USAGE

If the `—a` option is not used, non-directories given as arguments are not listed.

Files with holes in them may get an incorrect (high) estimate.

Currently not all systems report in terms of 512-byte units. This situation may change in the future.

## CHANGE HISTORY

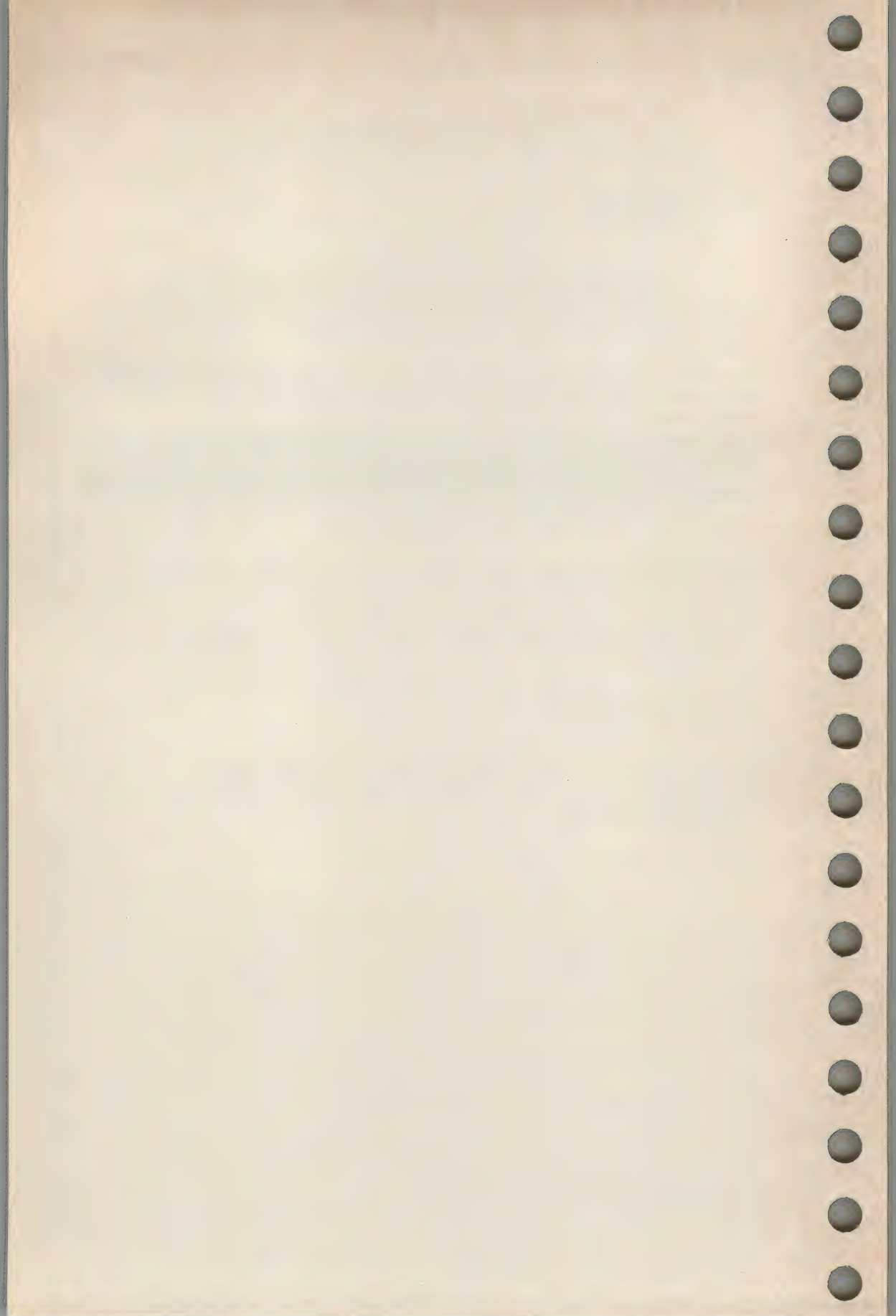
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The paragraph describing the `—r` option has been changed to reflect the behaviour of differing implementations.





## NAME

echo — echo arguments

## SYNOPSIS

echo [ arg ... ]

## DESCRIPTION

The command *echo* writes its *arguments* separated by blanks and terminated by a newline on the standard output. It also understands the following escape conventions:

- `\b` backspace
- `\c` print arguments up to this point, without newline; ignore remainder of command line
- `\f` form-feed
- `\n` newline
- `\r` carriage return
- `\t` tab
- `\v` vertical tab
- `\\` backslash
- `\On` *n* must be a 1-, 2- or 3-digit octal number; specifies the corresponding character

## SEE ALSO

sh(1).

## APPLICATION USAGE

The command *echo* is useful for producing diagnostics in command scripts and for sending known data into a pipe.

Arguments containing blanks and escape sequences must be suitably quoted.

This is usually a shell built-in command.

## CHANGE HISTORY

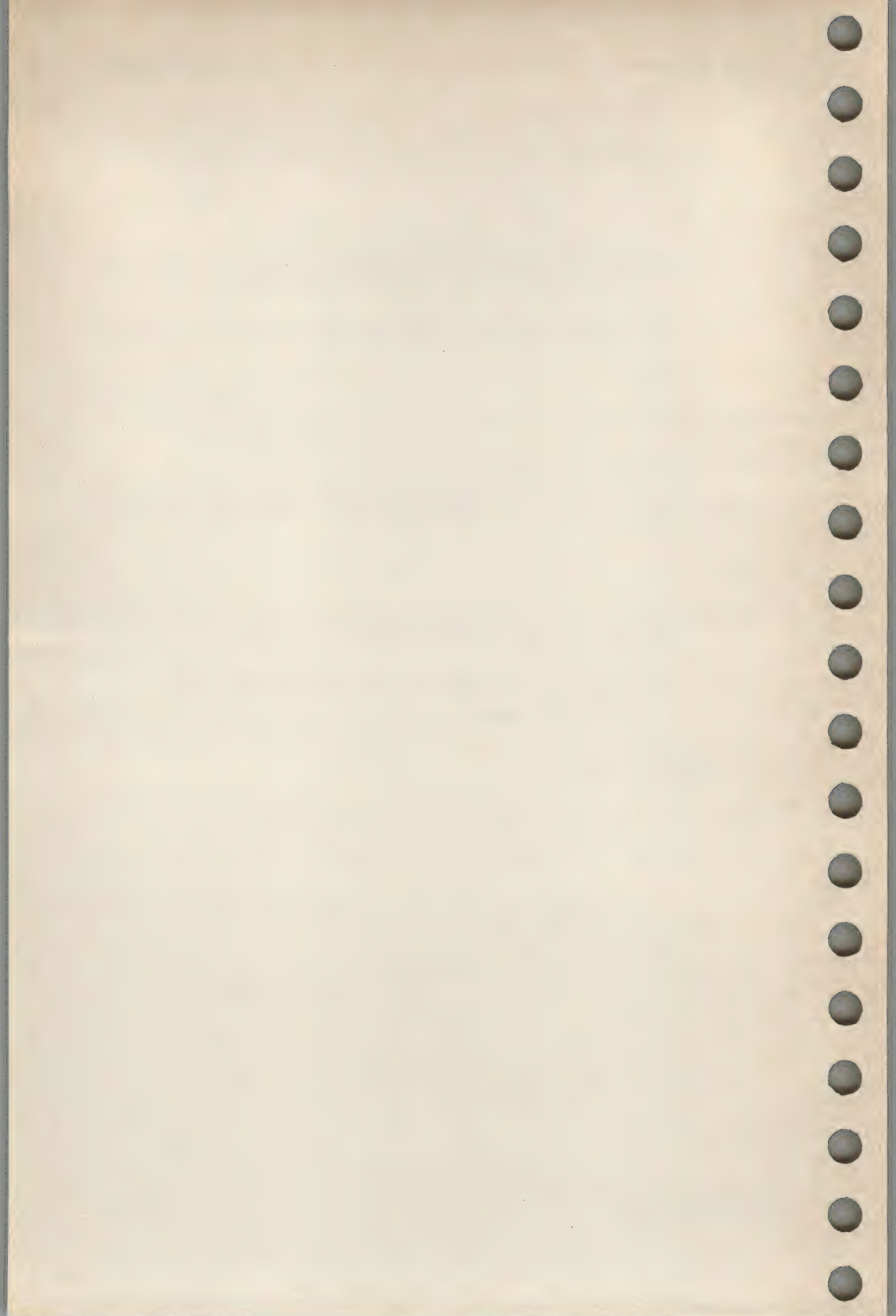
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The word "ASCII" has been removed from the description of the `\On` escape.





## NAME

ed, red — text editor

## SYNOPSIS

ed [—] [—p string] [file]

red [—] [—p string] [file]

## DESCRIPTION

The command *ed* is a text editor. If the *file* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into the *ed* buffer so that it can be edited.

The option *—* suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the *!* prompt after a *!command*.

The *—p* option allows the user to specify a prompt string.

*Ed* operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

The command *red* is a restricted version of *ed*. It will only allow editing of files in the current directory, and prohibits executing commands via *!command*. Attempts to bypass these restrictions result in an error message.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, no commands are recognised; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

*Ed* supports simple regular expression syntax.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; the exact effect on the current line is discussed under the description of each command. Addresses are constructed as follows:

1. The character *.* addresses the current line.
2. The character *\$* addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*th line of the buffer.
4. *x* addresses the line marked with the mark name character *x*, which must be a lower case letter. Lines are marked with the *k* command described below.



5. A regular expression (RE) enclosed by slashes (/) addresses the first line found by searching forward from the line following the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. The null RE (i.e., //) is equivalent to the last RE encountered.
6. A RE enclosed in question marks (?) addresses the first line found by searching backward from the line preceding the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line.
7. An address followed by a plus sign (+) or a minus sign (-) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with + or -, the addition or subtraction is taken with respect to the current line; e.g., -5 is understood to mean .-5.
9. If an address ends with + or -, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address - refers to the line preceding the current line. Moreover, trailing + and - characters have a cumulative effect, so -- refers to the current line less 2.
10. For convenience, a comma (,) stands for the address pair 1,\$, while a semicolon (;) stands for the pair .,\$.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (,). They may also be separated by a semicolon (;). In the latter case, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are not part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by *l*, *n*, or *p* in which case the current line is either listed, numbered or printed, respectively, as discussed below under the *l*, *n*, *p* commands.

(.)a  
<text>

The append command reads the given *text* and appends it after the addressed line; the current line becomes the last inserted line, or, if there were none, the addressed line. Address 0 is legal for this command: it causes the "appended" text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c  
<text>

The change command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

(...)d

The delete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

**e file** The edit command causes the entire contents of the buffer to be deleted, and then the named *file* to be read in; . is set to the last line of the buffer. If no file name is given, the currently-remembered file name, if any, is used (see the *f* command). The number of characters read is typed; the name *file* is remembered for possible use as a default file name in subsequent *e*, *r*, *w* commands. If *file* is replaced by *!*, the rest of the line is taken to be a command to the command interpreter whose output is to be read. Such a command is not remembered as the current file name.

**E file** The *E* (edit) command is like *e*, except that the editor does not check to see if any changes have been made to the buffer since the last *w* command.

**f file** If *file* is given, the file-name command changes the currently-remembered file name to *file*; otherwise, it prints the currently-remembered file name.

(1,\$)g/RE/command list

In the global command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with . initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; *a*, *i*, and *c* commands and associated input are permitted. The . terminating input mode may be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the *p* command. The *g*, *G*, *v*, and *V* commands are not permitted in the *command list*.

(1,\$)G/RE/

In the interactive global command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, . is changed to that line, and any one command (other than one of the *a*, *c*, *i*, *g*, *G*, *v*, and *V* commands) may be input and is executed. After the execution of



that command, the next marked line is printed, and so on; a newline acts as a null command; an `&` causes the re-execution of the most recent command executed within the current invocation of `G`. Note that the commands input as part of the execution of the `G` command may address and affect any lines in the buffer. The `G` command can be terminated by sending an interrupt signal.

- LA **h** The help command gives a short error message that explains the reason for the most recent `?` diagnostic.
- LA **H** The help command causes `ed` to enter a mode in which error messages are printed for all subsequent `?` diagnostics. It will also explain the previous `?` if there was one. The `H` command alternately turns this mode on and off; it is initially off.

(**.**)**i**  
`<text>`

The insert command inserts the given `text` before the addressed line; `.` is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the `a` command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(**.**,**+**)**j**  
 The join command joins contiguous lines by removing the appropriate newline characters. If exactly one address is given, this command does nothing.

(**.**)**kx**  
 The mark command marks the addressed line with name `x`, which must be a lower case letter. The address `ˆx` then addresses this line; `.` is unchanged.

OF (**...**)**l**  
 The list command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., `<tab>`, `<backspace>`) are represented by (hopefully) mnemonic overstrikes. All other non-printing characters are printed in octal, and long lines are folded. An `l` command may be appended to any other command other than `e`, `f`, `r`, or `w`.

(**...**)**ma**  
 The move command repositions the addressed line(s) after the line addressed by `a`. Address 0 is legal for `a` and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address `a` falls within the range of moved lines; `.` is left at the last line moved.

(**...**)**n**  
 The number command prints the addressed lines, preceding each line by its line number and a tab character; `.` is left at the last line printed. The `n` command may be appended to any other command other than `e`, `f`, `r` or `w`.

(**...**)**p**  
 The print command prints the addressed lines; `.` is left at the last line printed. The `p` command may be appended to any other command other than `e`, `f`, `r` or

w. For example, *dp* deletes the current line and prints the new current line.

- P The editor will prompt with a \* for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.
- q The quit command causes *ed* to exit.
- Q The editor exits without checking if changes have been made in the buffer since the last *w* command.

(*\$*)*r* *file*

The read command reads in the given *file* after the addressed line. If no file name is given, the currently-remembered file name, if any, is used (see *e* and *f* commands). The currently-remembered file name is not changed unless *file* is the very first file name mentioned since *ed* was invoked. Address 0 is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; . is set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a command to the command interpreter whose output is to be read. Such a command is not remembered as the current file name.

- (...)*s*/RE/replacement/ or
- (...)*s*/RE/replacement/*g* or
- (...)*s*/RE/replacement/*n*    *n* = 1-512

The substitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) matched strings are replaced by the *replacement* if the global replacement indicator *g* appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. If a number *n* appears after the command, only the *n*th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or newline may be used instead of / to delimit the RE and the *replacement*; . is left at the last line on which a substitution occurred.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of & in this context may be suppressed by preceding it by \. As a more general feature, the characters \n, where *n* is a digit, are replaced by the text matched by the *n*th regular subexpression of the specified RE enclosed between \( and \). When nested parenthesised subexpressions are present, *n* is determined by counting occurrences of \( starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

A line may be split by substituting a newline character into it. The newline in the *replacement* must be escaped by preceding it by \. Such substitution cannot be done as part of a *g* or *v* command list.



(...)*ta*

This command acts just like the *m* command, except that a *copy* of the addressed lines is placed after address *a* (which may be 0); *.* is left at the last line of the copy.

*u*

The undo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent *a*, *c*, *d*, *g*, *i*, *j*, *m*, *r*, *s*, *t*, *v*, *G*, or *V* command.

(1,\$)*N*/RE/command list

This command is the same as the global command *g* except that the *command list* is executed with *.* initially set to every line that does not match the RE.

(1,\$)*V*/RE/

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do not match the RE.

(1,\$)*w* file

The write command writes the addressed lines into the named file. The currently-remembered file name is not changed unless *file* is the very first file name mentioned since *ed* was invoked. If no file name is given, the currently-remembered file name, if any, is used (see *e* and *f* commands); *.* is unchanged. If the command is successful, the number of characters written is typed. The file is created if it does not exist. If *file* is replaced by *!*, the rest of the line is taken to be a command to the command interpreter whose standard input is the addressed lines. Such a command is not remembered as the current file name.

(\$)=

The line number of the addressed line is typed; *.* is unchanged by this command.

!command

The remainder of the line after the *!* is sent to the command interpreter to be interpreted as a command. Within the text of that *command*, the unescaped character *%* is replaced with the remembered file name; if a *!* appears as the first character of the command, it is replaced with the text of the previous command. Thus, *!!* will repeat the last command. If any expansion is performed, the expanded line is echoed; *.* is unchanged.

(.+1)

An address alone on a line causes the addressed line to be printed. A newline alone is equivalent to *.+1p*; it is useful for stepping forward through the buffer.

If an interrupt signal is sent, *ed* prints a *?* and returns to its command level.

If the closing delimiter of a RE or of a replacement string (e.g., */*) would be the last character before a newline, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

```
s/s1/s2  s/s1/s2/p
g/s1     g/s1/p
?s1      ?s1?
```

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy the editor buffer via the *e* or *q* commands. It prints *?* and allows the user to continue editing. A second *e* or *q* command at this point will take effect. The *—* command-line option inhibits this feature.

#### FILES

*ed.hup* work is saved here if *ed* receives the SIGHUP signal.

#### ERRORS

*?* for command errors.  
*?file* for an inaccessible file.

Use the *h* and *H* (help) commands for detailed explanations.

#### SEE ALSO

*sed*(1), *sh*(1).

#### APPLICATION USAGE

A *!* command cannot be subject to a *g* or a *v* command.

The sequence *\n* in a RE does not match a newline character.

If the editor input is coming from a command file (i.e., *ed file <ed-cmd-file*), the editor will exit at the first failure of a command that is in the command file.

#### FUTURE DIRECTIONS

The option *—* will be replaced by *—s*, in order to conform to the syntax standard. The old form of the option will continue to be accepted for some time.

#### CHANGE HISTORY

First released in Issue 2.

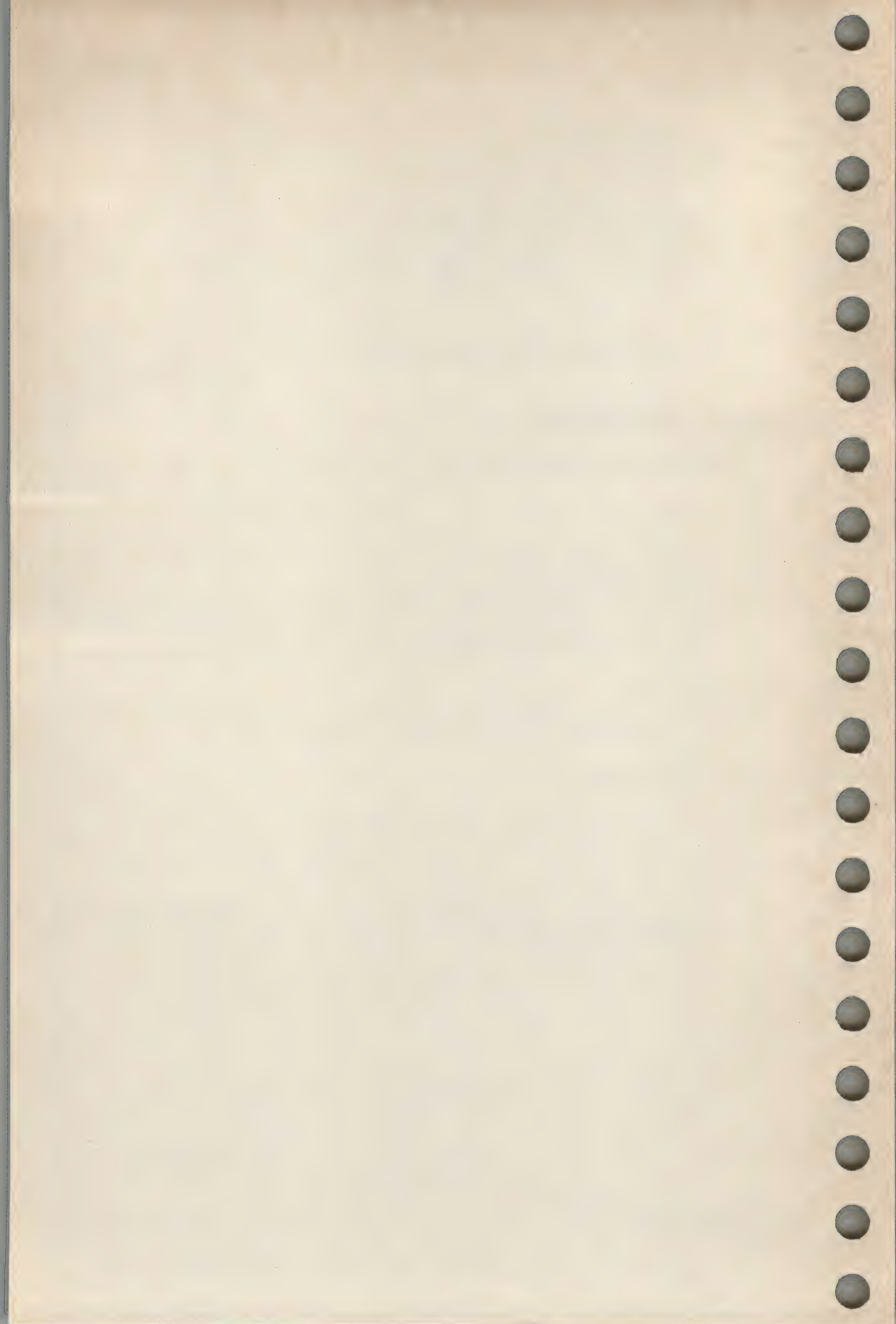
#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

In the descriptions of the *e*, *r* and *w* commands, the words "to the command interpreter" have been added to the explanation of the *!* modifier.

The sentence "The file is created if it does not exist." has been added to the description of the *w* command.





## NAME

egrep, fgrep — search a file for a pattern

## SYNOPSIS

```
egrep [ options ] [ expression ] [ file ... ]
```

LV

```
fgrep [ options ] [ string ... ] [ file ... ]
```

LV

## DESCRIPTION

The *egrep* and *fgrep* commands search the input *file* or files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The patterns used by *egrep* are extended regular expressions; *fgrep* patterns are fixed strings. The following *options* are recognised:

- v All lines but those matching are printed.
- x (Exact) Only lines matched in their entirety are printed (*fgrep* only).
- c Only a count of matching lines is printed.
- i Ignore upper/lower case distinction during comparisons.
- l Only the names of files with matching lines are listed (once), separated by newlines.
- n Each line is preceded by its relative line number in the file.
- e *expression*  
(*Egrep* only.) Same as a simple expression argument, but useful when the *expression* begins with a —.
- f *file*

The regular expression (*egrep*) or strings list (*fgrep*) is taken from the *file*.

In all cases, the file name is output if there is more than one input file. Care should be taken when using characters in *expression* that may also be meaningful to the command interpreter. When using *sh*(1), it is safest to enclose the entire *expression* argument in single quotes `...'`.

**fgrep**

*Fgrep* searches for lines that contain one of the *strings* separated by newlines.

**egrep**

*Egrep* accepts extended regular expression syntax, with the additional feature that newline is also a regular expression alternation character.

## EXIT STATUS

Exit status is:

- 0 matches are found
- 1 no matches found
- 2 syntax errors or inaccessible files (even if matches found in other files)

## SEE ALSO

grep(1), sed(1), stdio(5).

## APPLICATION USAGE

Lines are limited to BUFSIZ characters; longer lines are truncated. (BUFSIZ is defined in *stdio*(5).)



## FUTURE DIRECTIONS

The functionality of *egrep* and *fgrep* will eventually be provided in *grep*(1) and these two commands discontinued.

## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID where it is marked "Level 2: December 1, 1985".

The following change has been applied:

The description of the regular expressions used has been replaced by the paragraph "*Egrep* accepts extended regular expression syntax, with the additional feature that newline is also an alternator."

## NAME

`env` — set environment for command execution

## SYNOPSIS

`env [ — ] [[ name=value ] ... ] [ command [ arg ... ] ]`

## DESCRIPTION

The command *env* obtains the current environment, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* modify the execution environment: they are merged into the inherited environment before the command is executed. The `—` option causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

Optionally, arguments may be passed to the command by placing them as separate words after the command name.

If no command is specified, the resulting environment is printed, one name-value pair per line.

## SEE ALSO

`sh(1)`.

## CHANGE HISTORY

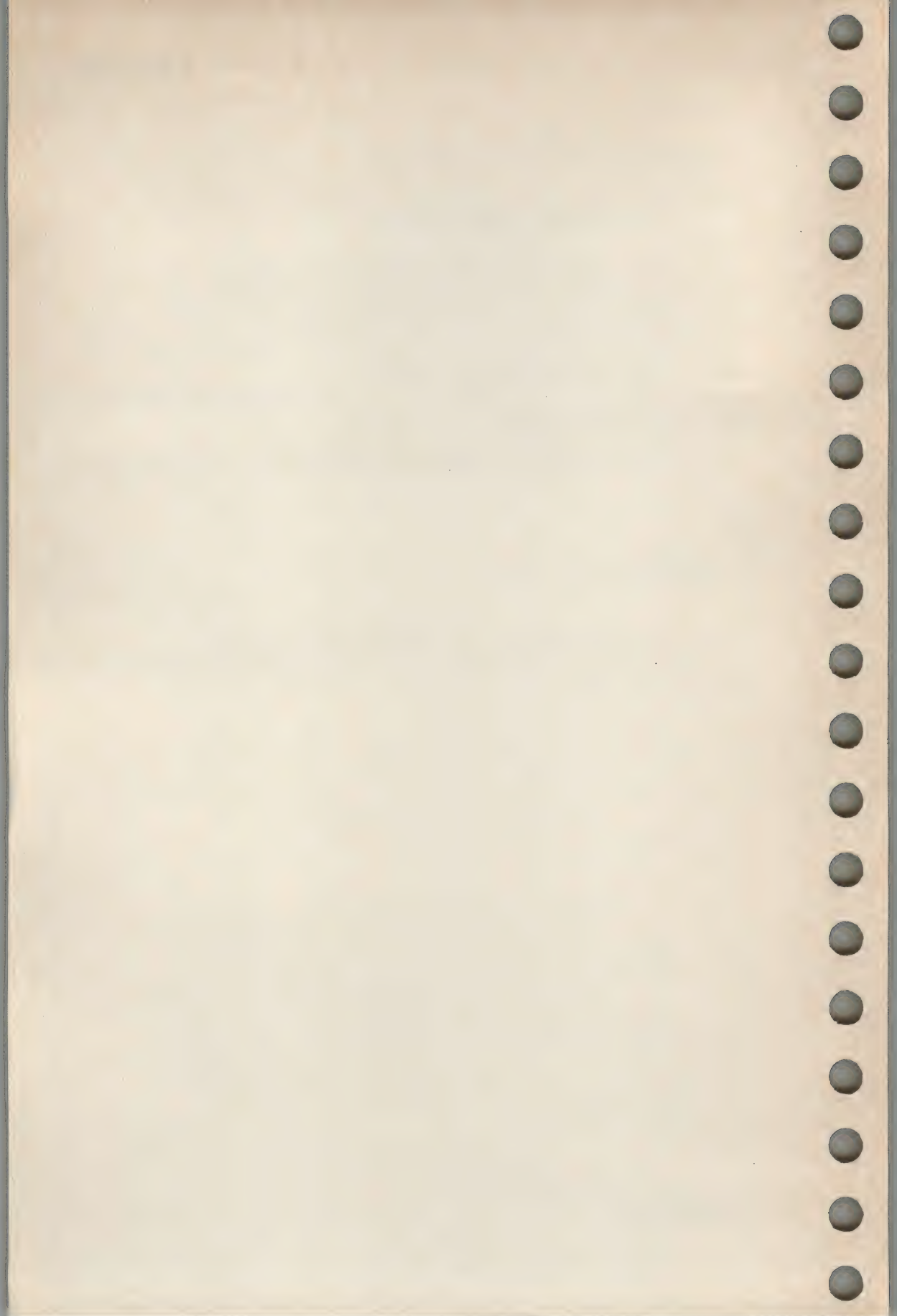
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The paragraph indicating that arguments may be passed to the command has been added.





## NAME

ex — text editor

## SYNOPSIS

ex [**—**] [**—v**] [**—r**] [**—R**] [**+command**] [**—l**] [*file* ...]

OP

## DESCRIPTION

The command *ex* is a line oriented text editor, which supports both command and display editing, see *vi*(1). The command line options are:

- Suppress all interactive-user feedback. This is useful in processing editor scripts.
- v** Invokes *vi*.
- r** Recover *file* or files after an editor or system crash. If no *file* is specified a list of all saved files will be printed.
- R** *Readonly* mode set, prevents accidentally overwriting the file.
- +command**  
Begin editing by executing the specified editor search or positioning *command*.
- l** *Lisp* mode; indents appropriately for *lisp* code; the *() {} [[ and ]]* commands in *vi* are modified to have meaning for *lisp*.

The *file* argument(s) indicates files to be edited, in the order specified.

The name of the file being edited by *ex* is the *current* file. The text of the file is read into a buffer and all editing changes are performed in this buffer; changes have no effect on the file until the buffer is written out explicitly.

The *alternate* file name is the name of the last file mentioned in an editor command, or the previous current file name if the last file mentioned became the current file. The character *%* in filenames is replaced by the current file name, and the character *#* by the alternate file name.

The named buffers *a* through *z* may be used for saving blocks of text during the edit. If the buffer name is specified in upper case, the buffer is appended to rather than being overwritten.

The *readonly* mode can be cleared from within the edit by setting the *noreadonly* edit option. Writing to a different file is allowed in *readonly* mode; in addition, the write can be forced by using *!* (see the *write* command below).

When an error occurs *ex* sends the BEL character to the terminal (to sound the bell) and prints a message. If an interrupt signal is received, *ex* returns to the command level in addition to the above actions. If the editor input is from a file, *ex* exits at the interrupt.

If *ex* detects an internal error, it attempts to preserve the buffer if any unwritten changes were made. The command line option **—r** is used to retrieve the saved changes.



At the beginning, *ex* is in the *command* mode, which is indicated by the `:` prompt. The *input* mode is entered by *append*, *insert*, or *change* commands; it is left (and *command* mode re-entered) by typing a period `.` alone at the beginning of a line.

Command lines beginning with the double quote character (`"`) are ignored. (This may be used for comments in an editor script.)

### Addressing

Dot (`.`) refers to the current line. There is always a current line; the positioning may be the result of an explicit movement by the user, or the result of a command that affected multiple lines (in which case it is usually the last line affected).

- `n` The *n*th line in the buffer, with lines numbered sequentially from 1.
- `$` The last line in the buffer.
- `%` Abbreviation for `1,$`; the entire buffer.
- `+n`
- `—n` An offset relative to the current line. (The forms `.+3`, `+3`, and `+++` are equivalent.)
- `/re/`
- `?re?` Line containing the pattern (see **Regular Expressions**, below) *re*, scanning forward (`/`) or backward (`?`). The trailing `/` or `?` may be omitted if the line is only to be printed. If the pattern is omitted, the previous pattern specified is used.
- `x` Lines may be marked using single lower case letters (see the *mark* command below); `x` refers to line marked *x*. In addition, the previous current line is marked before each non-relative motion; this line may be referred to by using `'` for *x*.

Addresses to commands consist of a series of line addresses (specified as above), separated by `,` or `;`. Such address lists are evaluated left-to-right. When `;` is the separator, the current line is set to the value of the previous address before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. Where a command requires two addresses, the first line must precede the second one in the buffer. A null address in a list defaults to the current line.

## Command Names and Abbreviations

abbrev	<i>ab</i>	next	<i>n</i>	unmap	<i>unm</i>
append	<i>a</i>	number	<i>≠ nu</i>	version	<i>ve</i>
args	<i>ar</i>	preserve	<i>pre</i>	visual	<i>vi</i>
change	<i>c</i>	print	<i>p</i>	write	<i>w</i>
copy	<i>co</i>	put	<i>pu</i>	xit	<i>x</i>
delete	<i>d</i>	quit	<i>q</i>	yank	<i>ya</i>
edit	<i>e</i>	read	<i>re</i>	(window)	<i>z</i>
file	<i>f</i>	recover	<i>re</i>	(escape)	<i>!</i>
global	<i>g v</i>	rewind	<i>rew</i>	(lshift)	<i>&lt;</i>
insert	<i>i</i>	set	<i>se</i>	(rshift)	<i>&gt;</i>
join	<i>j</i>	shell	<i>sh</i>	(resubst)	<i>&amp; s</i>
list	<i>l</i>	source	<i>so</i>	(scroll)	<i>^D</i>
map		substitute	<i>s</i>	(line no)	<i>=</i>
mark	<i>k ma</i>	unabbrev	<i>una</i>		
move	<i>m</i>	undo	<i>u</i>		

## Command Descriptions

In the following, *line* is a single line address, given in any of the forms described in the **Addressing** section above; *range* is a pair of line addresses, separated by a comma or semicolon (see the **Addressing** section for the difference between the two); *count* is a positive integer, specifying the number of lines to be affected by the command; *flags* is one or more of the characters *≠*, *p*, and *l*; the corresponding command to print the line is executed after the command completes. Any number of *+* or *-* characters may also be given with these flags.

When *count* is used, *range* is not effective; only a line number should be specified instead, to indicate the first line affected by the command. (If a range is given, then the last line of the range is taken as the starting line for the command.)

These modifiers are all optional; the defaults are as follows, unless otherwise stated: the default for *line* is the current line; the default for *range* is the current line only (*..*); the default for *count* is 1; the default for *flags* is null.

When only a *line* or a *range* is specified (with a null command), the implied command is *print*; if a null line is entered, the next line is printed (equivalent to *.+ 1p*)

**ab** *word rhs*

Add the named abbreviation to the current list. In visual mode, if *word* is typed as a complete word during input, it is replaced by the string *rhs*.

**line a** Enters *input* mode; the input text is placed after the specified line. If line 0 is specified, the text is placed at the beginning of the buffer. The last input line becomes the current line, or the target line, if no lines are input.

**ar** The argument list is printed, with the current argument inside *[* and *]*.

**range c count**

Enters *input* mode; the input text replaces the specified lines. The last input line becomes the current line; if no lines are input, the effect is the same as a



delete.

*range* **co** *line flags*

A copy of the specified lines (*range*) is placed after the specified destination line; line 0 specifies that the lines are to be placed at the beginning of the buffer.

*range* **d** *buffer count*

The specified lines are deleted from the buffer. If a named *buffer* is specified, the deleted text is saved in it. The line after the deleted lines becomes the current line, or the last line if the deleted lines were at the end.

**e**[ *+ line* ] *file*

Begin editing a new file. If the current buffer has been modified since the last write, then a warning is printed and the command is aborted. This action may be overridden by appending the character **!** to the command (*e! file*). The current line is the last line of the buffer; however, if this command is executed from within *visual*, the current line is the first line of the buffer. If the *+ line* option is specified, the current line is set to the specified position, where *line* may be a number (or **\$**) or specified as */re* or *?re*.

**f** Prints the current file name and other information, including the number of lines and the current position.

*range* **g** */re/ command ...*

First marks the lines within the given *range* that match the given pattern. Then the given *command* or commands are executed with **.** set to each marked line.

*Cmds* may be specified on multiple lines by hiding newlines with a backslash. If *cmds* are omitted, each line is printed. *Append*, *change*, and *insert* commands are omitted; the terminating dot may be omitted if it ends *cmds*. *Visual* commands are also permitted, and take input from the terminal.

The *global* command itself, and the *undo* command, are not allowed in *cmds*. The edit options *autoprint*, *autoindent* and *report* are inhibited.

*range* **v** */re/ command ...*

This is the same as the *global* command, except that *command* is run on the lines that do not match the pattern.

*line* **i**

Enters *input* mode; the input text is placed before the specified line. The last line input becomes the current line, or the line before the target line, if no lines were input.

*range* **j** *count flags*

Joins the text from the specified lines together into one line. White space is adjusted to provide at least one blank character, two if there was a period at the end of the line, or none if the first following character is a **)**. Extra white space at the start of a line is discarded.

Appending the command with a **!** causes a simpler join with no white space processing.

**range l count flags**

Prints the specified lines with tabs printed as ^I and the end of each line marked with a trailing \$. (The only useful flag is # for line numbers.) The last line printed becomes the current line.

**map x rhs**

The *map* command is used to define macros for use in *visual* mode. The first argument is a single character, or the sequence #*n*, where *n* is a digit, to refer to the function key *n*. When this character or function key is typed in *visual* mode, the action is as if the corresponding *rhs* had been typed. If ! is appended to the command *map*, then the mapping is effective during insert mode rather than command mode. Special characters, white space, and newline must be escaped with a ^V to be entered in the arguments.

**line ma x**

(The letter *k* is an alternative abbreviation for the *mark* command.) The specified line is given the specified mark *x*, which must be a single lower case letter. (The *x* must be preceded by a space or tab.) The current line position is not affected.

**range m line**

Moves the specified lines (*range*) to be after the target line. The first of the moved lines becomes the current line.

- n** The next file from the command line argument list is edited. Appending a ! to the command overrides the warning about the buffer having been modified since the last write (discarding any changes). The argument list may be replaced by specifying a new one on this command line.

**range nu count flags**

(The character # is an alternative abbreviation for the *number* command.) Prints the lines, each preceded by its line number. (The only useful flag is !.) The last line printed becomes the current line.

- pre** The current editor buffer is saved as though the system had just crashed. This command is for use in emergencies, for example when a write does not work, and the buffer cannot be saved in any other way.

**range p count**

Prints the specified lines, with non-printing characters printed as control characters in the form ^*x*; DEL is represented as ^?. The last line printed becomes the current line.

**line pu buffer**

Puts back deleted or "yanked" lines. A buffer may be specified; otherwise, the text in the unnamed buffer (where deleted or yanked text is placed by default) is restored.

- q** Causes termination of the edit. If the buffer has been modified since the last write, a warning is printed and the command fails. This warning may be overridden by appending a ! to the command (discarding changes).



**line r file**

Places a copy of the specified *file* in the buffer after the target line (which may be line 0 to place text at the beginning). If no *file* is named the current file is the default. If there is no current file then *file* becomes the current file. The last line read becomes the current line; in *visual* the first line read becomes the current line.

If *file* is given as *!string* then *string* is taken to be a system command, and passed to the command interpreter; the resultant output is read in to the buffer. A blank or tab must precede the *!*.

**rec file**

Recovers *file* from the save area, after an accidental hangup or a system crash.

**rew** The argument list is rewound, and the first file in the list is edited. Any warnings may be overridden by appending a *!*.

**se parameter**

With no arguments, the *set* command prints those options whose values have been changed from the default settings; with the parameter *all* it prints all of the option values.

Giving an option name followed by a *?* causes the current value of that option to be printed. The *?* is necessary only for Boolean valued options. Boolean options are given values by the form *se option* to turn them on, or *se no option* to turn them off; string and numeric options are assigned by the form *se option=value*. More than one parameter may be given; they are interpreted left to right.

See Edit Options below for further details about options.

**sh** The user is put into the command interpreter, usually *sh*, see *sh(1)*; editing is resumed on exit.

**so file**

Reads and executes commands from the specified *file*. Such *so* commands may be nested.

**range s /re/repl/ options count flags**

On each specified line, the first instance of the pattern *re* is replaced by the string *repl*. (See Regular Expressions and Replacement Strings below.) If *options* includes the letter *g* (global), then all instances of the pattern in the line are substituted. If the option letter *c* (confirm) is included, then before each substitution the line is typed with the pattern to be replaced marked with *^* characters; a response of *y* causes the substitution to be done, while any other input aborts it. The last line substituted becomes the current line.

**una word**

Delete *word* from the list of abbreviations.

**u** Reverses the changes made by the previous editing command. For this purpose, *global* and *visual* are considered single commands. Commands which affect the external environment, such as *write*, *edit* and *next*, cannot be undone.

An *undo* can itself be reversed.

**unm x**

The macro definition for *x* is removed.

MV

**ve** Prints the current version of the editor.

**line vi type count**

Enters *visual* mode at the specified *line*. The *type* is optional, and may be — or ., as in the *z* command, to specify the position of the specified line on the screen window. (The default is to place the line at the top of the screen window.) A *count* specifies an initial window size; the default is the value of the edit option *window*. The command *Q* exits *visual* mode. For more information, see *vi*(1).

**range w file**

Writes the specified lines (the whole buffer, if no *range* is given) out to *file*, printing the number of lines and characters written. If *file* is not specified, the default is the current file. (The command fails with an error message if there is no current file and no file is specified.)

If an alternate file is specified, and the file exists, then the write will fail; it may be forced by appending a *!* to the command. An existing file may be appended to by appending *>>* to the command. If the file does not exist, an error is reported.

If the file is specified as *!string*, then *string* is taken as a system command; the command interpreter is invoked, and the specified lines are passed as standard input to the command.

The command *wq* is equivalent to a *w* followed by a *q*; *wq!* is equivalent to *w!* followed by *q*.

**x** Writes out the buffer if any changes have been made, and then (in any case) quits.

**range ya buffer count**

Places the specified lines in the named *buffer*. If no buffer is specified, the unnamed buffer is used (where the most recently deleted or yanked text is placed by default).

**line z type count**

If *type* is omitted, then *count* lines following the specified *line* (default current line) are printed. The default for *count* is the value of the edit option *window*.

If *type* is specified, it must be — or .; a — causes the line to be placed at the bottom of the screen, while a . causes the line to be placed in the middle. The last line printed becomes the current line.

**! command**

The remainder of the line after the *!* is passed to the system command interpreter for execution. A warning is issued if the buffer has been changed since the last write. A single *!* is printed when the command completes. The



current line position is not affected.

Within the text of *command* % and # are expanded as filenames, and ! is replaced with the text of the previous ! command. (Thus !! repeats the previous ! command.) If any such expansion is done, the expanded line will be echoed.

#### *range!* *command*

In this form of the ! command, the specified lines (there is no default; see previous paragraph) are passed to the command interpreter as standard input; the resulting output replaces the specified lines.

#### *range* < *count*

Shift the specified lines to the left; the number of spaces to be shifted is determined by the edit option *shiftwidth*. Only white space (blanks and tabs) is lost in shifting; other characters are not affected. The last line changed becomes the current line.

#### *range* > *count*

Shift the specified lines to the right by inserting white space (see previous paragraph for further details).

#### *range* & *options* *count* *flags*

Repeats the previous *substitute* command, as if & were replaced by the previous *s/re/repl/*. (The same effect is obtained by omitting the */re/repl/* string in the *substitute* command.)

#### *^D* (control-D)

*^D* (ASCII EOT) prints the next *n* lines, where *n* is the value of the edit option *scroll*.

#### *line* =

Prints the line number of the specified *line* (default last line). The current line position is not affected.

### Regular Expressions

Regular expressions are interpreted according to the setting of the edit option *magic*; the following assumes the setting *magic*. The differences caused by the setting *nomagic* are described below.

*Ex* makes use of simple regular expression syntax, with the following additions:

#### *\<... \>*

The sequence *\<* matches the beginning of a "word". That is, the matched string must begin in a letter, digit, or underline, and be preceded by the beginning of the line or a character other than the above. The sequence *\>* matches the end of a "word".

Matches the replacement part of the last *substitute* command.

#### *\{... \}*

The use of braces to give a repetition count is not supported.

When *nomagic* is set, the only characters with special meanings are ^ at the beginning of a pattern, \$ at the end of a pattern, and \. The characters ., \*, [, and ~ lose their

special meanings, unless escaped by a `\`.

### Replacement Strings

The character `&` (`\&` if *nomagic* is set) in the replacement string stands for the text matched by the pattern to be replaced. The character `~` (`\~` if *nomagic* is set) is replaced by the replacement part of the previous *substitute* command. The sequence `\n`, where *n* is an integer, is replaced by the text matched by the pattern enclosed in the *n*th set of parentheses `\(` and `\)`. The sequence `\u` (`\U`) causes the immediately following character in the replacement to be converted to upper case (lower case), if this character is a letter. The sequence `\U` (`\L`) turns such conversion on, until the sequence `\E` or `\e` is encountered, or the end of the replacement string is reached.

### Edit Options

The command `ex` has a number of options that modify its behaviour. These options have default settings, which may be changed using the *set* command (see above). Options may also be set at startup by putting a *set* command string in the environment variable `EXINIT`, or in the file `.exrc` in the `HOME` directory, or in `.exrc` in the current directory.

Options are Boolean unless otherwise specified.

#### **autoindent, ai**

If *autoindent* is set, each line in *insert* mode is indented (using blanks and tabs) to align with the previous line. (Starting indentation is determined by the line appended after, or the line inserted before, or the first line changed.) Additional indentation can be provided as usual; succeeding lines will automatically be indented to the new alignment. Reducing the indent is achieved by typing `^D` one or more times; the cursor is moved back *shiftwidth* spaces for each `^D`. (A `^` followed by a `^D` removes all indentation temporarily for the current line; a `0` followed by a `^D` removes all indentation.)

#### **autoprint, ap**

The current line is printed after each command that changes buffer text. (Autoprint is suppressed in globals.)

#### **autowrite, aw**

The buffer is written (to the current file) if it has been modified, and a *next*, *rewind*, *!* command is given.

#### **beautify, bf**

Causes all control characters other than tab, newline and formfeed to be discarded from the input text.

#### **directory, dir**

The value of this option specifies the directory in which the editor buffer is to be placed. If this directory is not writable by the user, the editor quits.

#### **edcompatible, ed**

Causes the presence of *g* and *c* suffixes on substitute commands to be remembered, and toggled by repeating the suffixes.



**ignorecase, ic**

All upper case characters in the text are mapped to lower case in regular expression matching. Also, all upper case characters in regular expressions are mapped to lower case, except in character class specifications.

**lisp** *Autoindent* mode, and the ( ) { } [[ ]] commands in *visual* are suitably modified for *lisp* code.

**list** All printed lines will be displayed with tabs shown as ^I, and the end of line marked by a \$.

**magic**

Changes interpretation of characters in regular expressions and substitution replacement strings (see the relevant sections above).

**number, nu**

Causes lines to be printed with line numbers.

**paragraphs, para**

The value of this option is a string, in which successive pairs of characters specify the names of text-processing macros which begin paragraphs. (A macro appears in the text in the form .XX, where the . is the first character in the line.)

**prompt**

When set, command mode input is prompted for with a :; when unset, no prompt is displayed.

**redraw**

The editor simulates an intelligent terminal on a dumb terminal. (Since this is likely to require a large amount of output to the terminal, it is useful only at high transmission speeds.)

**remap**

If set, then macro translation allows for macros defined in terms of other macros; translation continues until the final product is obtained. If unset, then a one-step translation only is done.

**report**

The value of this option gives the number of lines that must be changed by a command before a report is generated on the number of lines affected.

**scroll**

The value of this option determines the number of lines scrolled on a ^D, and the number of lines displayed by the z command (twice the value of scroll).

**sections**

The value of this option is a string, in which successive pairs of characters specify the names of text-processing macros which begin sections. (See *paragraphs* option above.)

**shiftwidth, sw**

The value of this option gives the width of a software tab stop, used during

*autoindent*, and by the shift commands.

**showmatch, sm**

In *visual* mode, when a `)` or `}` is typed, the matching `(` or `{` is shown if it is still on the screen.

**slowopen, slow**

In *visual* mode, prevents screen updates during input to improve throughput on unintelligent terminals.

**tabstop, ts**

The value of this option specifies the software tab stops to be used by the editor to expand tabs in the input file.

**terse**

When set, error messages are shorter.

**window**

The number of lines in a text window in *visual* mode.

**wrapscan, ws**

When set, searches (using `//` or `??`) wrap around the end of the file; when unset, searches stop at the beginning or the end of the file, as appropriate.

**wrapmargin, wm**

In *visual* mode, if the value of this option is greater than zero (say *n*), then a newline is automatically added to an input line, at a word boundary, so that lines end at least *n* spaces from the right margin of the terminal screen.

**writeany, wa**

Inhibits the checks otherwise made before write commands, allowing a write to any file (provided the system allows it).

**FILES**

<code>\$HOME/.exrc</code>	editor initialisation file
<code>./exrc</code>	editor initialisation file

**SEE ALSO**

`vi(1)`.

**APPLICATION USAGE**

The *undo* command causes all marks to be lost on lines that were changed and then restored.

The *z* command prints a number of logical rather than physical lines. More than a screenful of output may result if long lines are present.

Null characters are discarded in input files and cannot appear in resultant files.

On some systems, recovery of an edit with the `—r` option may only be possible if certain system-dependent actions were taken when the system was restarted.

As this editor depends on the *optional curses(3X)* package, applications should be aware that it may not be present, or may not function with certain classes of terminal.



## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The description of the `—r` option has been taken from *vi*.

A reference to disabling the use of the bell action has been deleted, as the mechanism for doing this is not described.

## NAME

*expr* — evaluate expression

## SYNOPSIS

*expr* expression

## DESCRIPTION

The command *expr* evaluates *expression* and writes the result on the standard output. Terms of the expression must be separated by blanks. Characters special to the command interpreter must be suitably escaped. Note that 0 is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign.

The operators are listed below. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols. The symbol *arg* represents an argument.

*arg* | *arg*

returns the first *arg* if it is neither null nor 0, otherwise returns the second *arg*.

*arg* & *arg*

returns the first *arg* if neither *arg* is null or 0, otherwise returns 0.

*arg* { =, >, >=, <, <=, != } *arg*

returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

*arg* { +, - } *arg*

Addition or subtraction of integer-valued arguments.

*arg* { \*, /, % } *arg*

Multiplication, division, or remainder of the integer-valued arguments.

*arg* : *arg*

The matching operator : compares the first argument with the second argument which must be a regular expression. Simple regular expression syntax is used, except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (0 on failure). Alternatively, the \{... \} pattern symbols can be used to return a portion of the first argument.

## EXAMPLES

1. The following example adds 1 to the variable *a*.

```
a=`expr $a + 1`
```

2. For *\$a* equal to either */usr/abc/file* or just *file*, the following example returns the last segment of a path name (i.e., *file*). Care should be taken when using / alone as an argument: *expr* will take it as the division operator.

```
expr $a : '.*' \\. '*' \'
```



3. A better representation of example 2: the addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

```
expr // $a : '.*/ \(.*/ \)
```

4. The example below returns the number of characters in \$VAR.

```
expr $VAR : '.*'
```

## EXIT STATUS

As a side effect of expression evaluation, *expr* returns the following exit values:

- 0 if the expression is neither null nor 0
- 1 if the expression is null or 0
- 2 for invalid expressions.

## SEE ALSO

*sh*(1).

## APPLICATION USAGE

After argument processing, see *sh*(1), *expr* cannot tell the difference between an operator and an operand except by the value. If \$a is =, the command:

```
expr $a = '='
```

looks like:

```
expr = = =
```

as the arguments are passed to *expr* (and they will all be taken as the = operator).

The following works:

```
expr X$a = X=
```

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

Excessive \ characters have been removed from the DESCRIPTION.

## NAME

*file* — determine file type

## SYNOPSIS

*file* [ *-f* [*file*] *file* ...

OF LA PI

## DESCRIPTION

The command *file* performs a series of tests on each specified *file* in an attempt to classify it. If it appears to be a text file, *file* examines an initial segment and makes a guess about its language. (The answer is not guaranteed to be correct.) If an argument is an executable file it is identified as such, and any other available information is reported.

UN

If the *-f* option is given, the next argument is taken to be a file containing the names of the files to be examined.

## APPLICATION USAGE

This utility uses heuristics to determine the filetype and its accuracy should therefore not be relied upon.

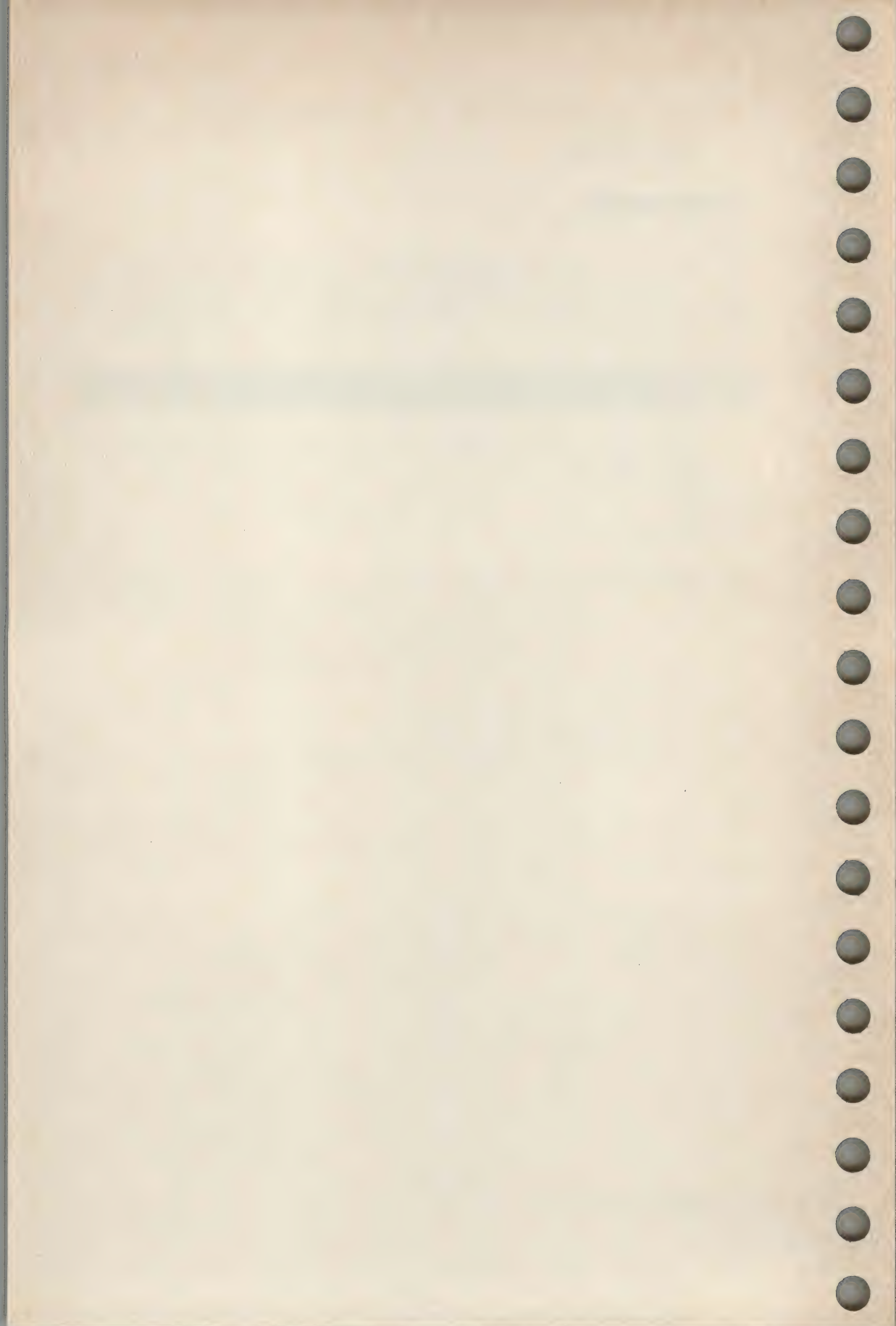
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

find — find files

## SYNOPSIS

find path-name ... expression

## DESCRIPTION

The command *find* recursively descends the directory hierarchy for each path name in the *path-name* list (i.e., one or more path names) seeking files that match a boolean *expression* written in the primaries given below. In the following descriptions, the argument *n* is used as a decimal integer where *+n* means more than *n*, *-n* means less than *n* and *n* means exactly *n*.

The *expression* argument is made up of:

—name *file*

True if *file* matches the current file name. Shell metanotation characters may be used in *file*.

—perm *onum*

True if the file permission flags exactly match the octal number *onum*, see *chmod*(1). If *onum* is prefixed by a minus sign, more flag bits (017777), see *stat*(2), become significant and the flags are compared.

—type *c*

UN

True if the type of the file is *c*, where *c* is *b*, *c*, *d*, *p* or *f* for block special file, character special file, directory, *file* (named pipe), or plain file respectively. Other types may exist, but should be avoided if portability is an issue.

—links *n*

True if the file has *n* links.

—user *uname*

True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the */etc/passwd* file, it is taken as a user ID.

—group *gname*

True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the */etc/group* file, it is taken as a group ID.

—size *n*[*c*]

True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a *c*, the size is in characters.

—atime *n*

True if the file has been accessed in *n* days. The access time of directories in the *path-name* list is changed by *find* itself.

—mtime *n*

True if the file has been modified in *n* days.

—ctime *n*

True if the file inode has been changed in *n* days.



—exec *cmd*

True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument { } is replaced by the current path name.

—ok *cmd*

Like —exec except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing y.

—print

Always true; causes the current path name to be printed.

—newer *file*

True if the current file has been modified more recently than the argument *file*.

—depth

Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*, see *cpio*(1), to transfer files that are contained in directories without write permission.

( *expression* )

True if the parenthesised expression is true (parentheses must be escaped if they are special to the command interpreter).

The primaries may be combined using the following operators (in order of decreasing precedence):

1. The negation of a primary (! is the unary *not* operator).
2. Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).
3. Alternation of primaries (—o is the *or* operator).

## EXAMPLE

To remove all files named *tmp* or ending in .xx that have not been accessed for a week:

```
find / \ ( —name tmp —o —name '*.xx' \ ) —atime +7 —exec rm { } \ ;
```

## FILES

/etc/passwd	system's password file
/etc/group	system's group file

## SEE ALSO

chmod(1), cpio(1), sh(1), test(1), stat(2).

## CHANGE HISTORY

First released in Issue 2.

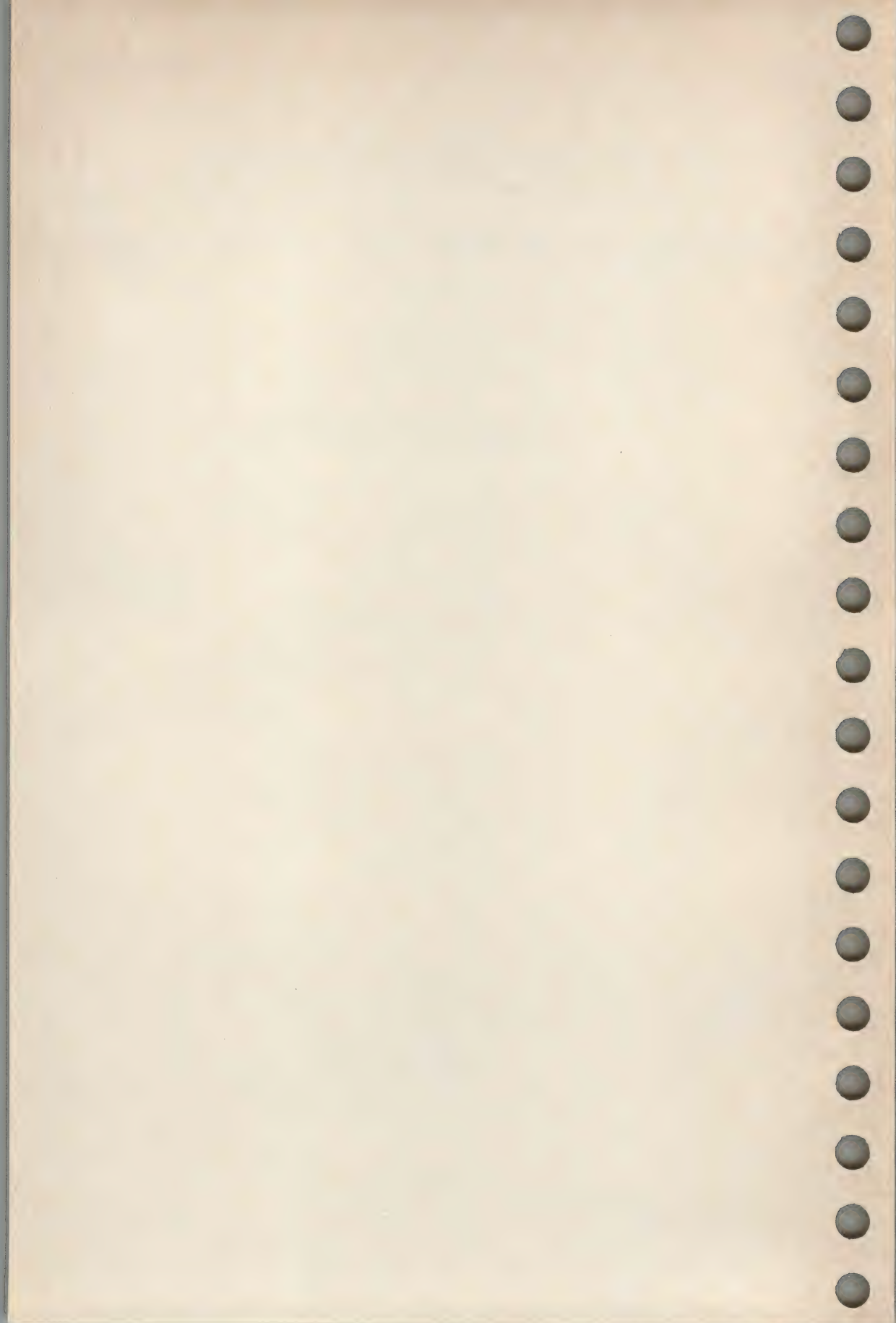
Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

A space has been added between the *perm*, *links* and *user* operators and their arguments.

In the description of the *—type* argument, the reference to the existence of other types has been added.





## NAME

get — get a version of an SCCS file

## SYNOPSIS

```
get [—rSID] [—ccutoff] [—e] [—b] [—ilist] [—xlist] [—k] [—l[p]] [—p]
    [—s] [—m] [—n] [—g] [—t] file...
```

## DESCRIPTION

The command *get* generates an ASCII text file from each named SCCS *file* according to the specifications given by its options, which begin with —. The arguments may be specified in any order, but all options apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If — is given as the name of a *file*, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading s. (see also FILES, below).

Each of the options is explained below as though only one SCCS file is to be processed, but the effects of any option apply independently to each named file.

—rSID The SCCS Identification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta* if the —e option is also used), as a function of the SID specified.

—ccutoff

Cutoff date-time, in the form:

```
YY[MM[DD[HH[MM[SS]]]]]
```

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, —c7502 is equivalent to —c750228235959.

Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: —c77/2/29:22:25

—e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*. The —e option used in a *get* for a particular version (SID) of the SCCS file prevents further *gets* for editing on the same SID until *delta* is executed or the j (joint edit) flag is set in the SCCS file. Concurrent use of *get* —e for different SIDs is always allowed.



If the *g-file* generated by *get* with a *—e* option is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the *—k* option in place of the *—e* option.

SCCS file protection specified via the ceiling, floor, and authorised user list stored in the SCCS file is enforced when the *—e* option is used.

- b Used with the *—e* option to indicate that the new delta should have an SID in a new branch as shown in Table 1. This option is ignored if the *b* flag is not present in the file or if the retrieved delta is not a leaf delta. (A leaf delta is one that has no successors on the SCCS file tree.)

**Note:** A branch delta may always be created from a non-leaf delta.

- i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

```
<list> ::= <range> | <list> , <range>
<range> ::= SID | SID - SID
```

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

- x*list*

A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the *—i* option for the *list* format.

- k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The *—k* option is implied by the *—e* option.

—l

- lp Causes a delta summary to be written into an *l-file*. If *—lp* is used then an *l-file* is not created; the delta summary is written on the standard output instead. See FILES for the format of the *l-file*.

- p Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to standard error instead, unless the *—s* option is used, in which case it disappears.

- s Suppresses all output normally written on the standard output. However, fatal error messages (which are always written to the standard error) remain unaffected.

- m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

- n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the —m and —n options are used, the format is: %M% value, followed by a horizontal tab, followed by the —m option generated format.
- g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.
- t Used to access the most recently created ("top") delta in a given release (e.g., —r1), or release and level (e.g., —r1.2).

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the —e option is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a newline) before it is processed. If the —i option is used included deltas are listed following the notation "Included"; if the —x option is used, excluded deltas are listed following the notation "Excluded".

SID★ Specified	-b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	-	R < mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	-	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk successor	R.L	R.(L+1)
R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	-	Trunk successor in release = R	R.L	R.L.(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.B.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.B.(mB+1).1
R.L.B.S	-	Branch successor	R.L.B.S	R.L.B.(mB+1).1

TABLE 1. Determination of SCCS Identification String



\* R, L, B, and S are the "release", "level", "branch", and "sequence" components of the SID, respectively; m means "maximum". Thus, for example, R.mL means "the maximum level number within release R"; R.L.(mB+1).1 means "the first sequence number on the new branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form R.L, R.L.B, or R.L.B.S, each of the specified components must exist.

\*\* hR is the highest existing release that is lower than the specified, nonexistent, release R.

\*\*\* This is used to force creation of the first delta in a new release.

† The *-b* option is effective only if the *b* flag is present in the file. An entry of - means "irrelevant".

‡ This case applies if the *d* (default SID) flag is not present in the file. If the *d* flag is present in the file, then the SID obtained from the *d* flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

#### Identification Keywords

Identifying information is inserted into the text retrieved from the SCCS file by replacing identification keywords with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	Value
%M%	Module name: either the value of the <i>m</i> flag in the file, or if absent, the name of the SCCS file with the leading <i>s.</i> removed.
%I%	SCCS identification (SID) (%R%.%L% or %R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the <i>t</i> flag in the SCCS file.
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the <i>q</i> flag in the file.
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this should not have happened" type errors. It is not intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognisable by <i>what</i> .
%W%	A shorthand notation for constructing <i>what</i> strings.

%W% = %Z%%M%<horizontal-tab>%I%

%A%      Another shorthand notation for constructing *what*(1D) strings.

%A% = %Z%%Y% %M% %I%%Z%

## FILES

Several auxiliary files may be created by *get*. These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form *s.module-name*, the auxiliary files are named by replacing the leading *s* with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the *s*. prefix. For example, *s.xyz.c*, the auxiliary file names would be *xyz.c*, *l.xyz.c*, *p.xyz.c*, and *z.xyz.c*, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the *—p* option is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. It is owned by the real user. If the *—k* option is used or implied it is writable by the owner only (read-only for everyone else); otherwise it is read-only. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the *—l* option is used; it is read-only and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

- a. A blank character if the delta was applied; \* otherwise.
- b. A blank character if the delta was applied or was not applied and ignored; \* if the delta was not applied and was not ignored.
- c. A code indicating a "special" reason why the delta was or was not applied:
  - I:    Included.
  - X:    Excluded.
  - C:    Cut off (by a *—c* option).
- d. Blank.
- e. SCCS identification (SID).
- f. Tab character.
- g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
- h. Blank.
- i. Login name of person who created *delta*.

The comments and *MR* data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with a *—e* option along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with a *—e* option for the same SID until *delta* is executed or the joint edit flag, *j*, is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. It is writable by owner only, and it is owned by the effective user. The format of the *p-file* is: the *g-file* SID, followed by a



blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the *—i* option if it was present, followed by a blank and the *—x* option if it was present, followed by a newline. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a lock-out mechanism against simultaneous updates. Its contents are the binary process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created read-only.

#### SEE ALSO

admin(1D), delta(1D), prs(1D), what(1D).

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The example *%R%.%L%* has been added to the description of the *%/%* identification keyword.

## NAME

grep — search a file for a pattern

## SYNOPSIS

grep [ options ] expression [ file ... ]

## DESCRIPTION

The command *grep* searches the input *file* or files (standard input default) for lines matching the pattern *expression*. Normally, each line found is copied to the standard output. Simple regular expression syntax is used in patterns.

The following *options* are recognised:

- v All lines but those matching are printed.
- c Only a count of matching lines is printed.
- i Ignore upper/lower case distinction during comparisons.
- l Only the names of files with matching lines are listed (once), separated by newlines.
- n Each line is preceded by its relative line number in the file.
- s The error messages produced for nonexistent or unreadable files are suppressed.

In all cases, the file name is output if there is more than one input file.

## EXIT STATUS

Exit status is:

- 0 matches are found
- 1 none found
- 2 syntax errors or inaccessible files (even if matches found in other files)

## SEE ALSO

egrep(1), sed(1).

## APPLICATION USAGE

Care should be taken when using characters in *expression* that may also be meaningful to the command interpreter. It is safest to enclose the entire *expression* argument in single quotes: `'...'`.

## FUTURE DIRECTIONS

The functionality of *egrep* and *fgrep*, see *egrep*(1) will eventually be provided in *grep*, and those two commands discontinued.

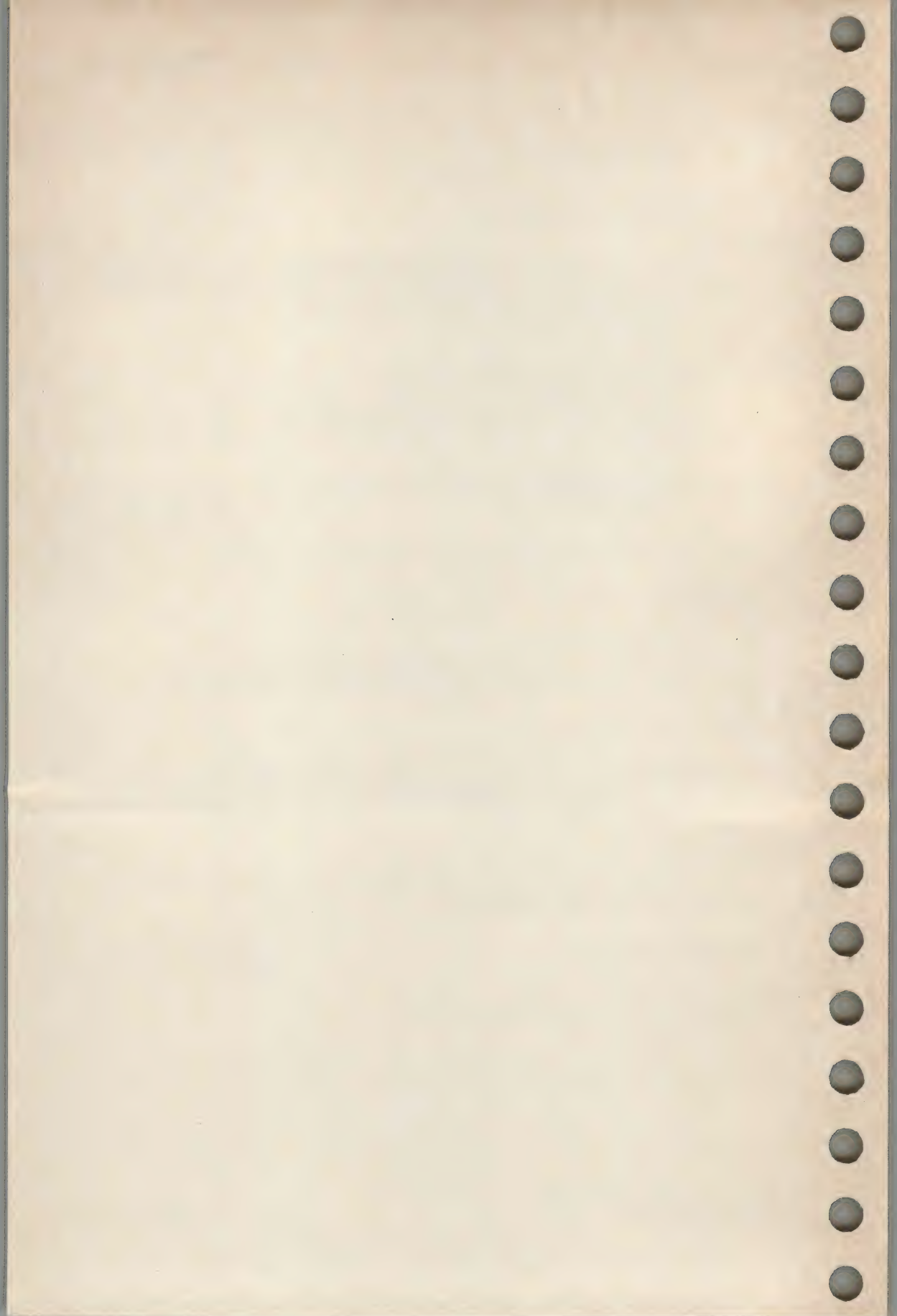
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`id` — print user and group IDs and names

## SYNOPSIS

`id`

OF

## DESCRIPTION

The command `id` writes a message on the standard output giving the user and group IDs and the corresponding names of the invoking process. If the effective and real IDs do not match, both are written.

## SEE ALSO

`logname(1)`, `getuid(2)`.

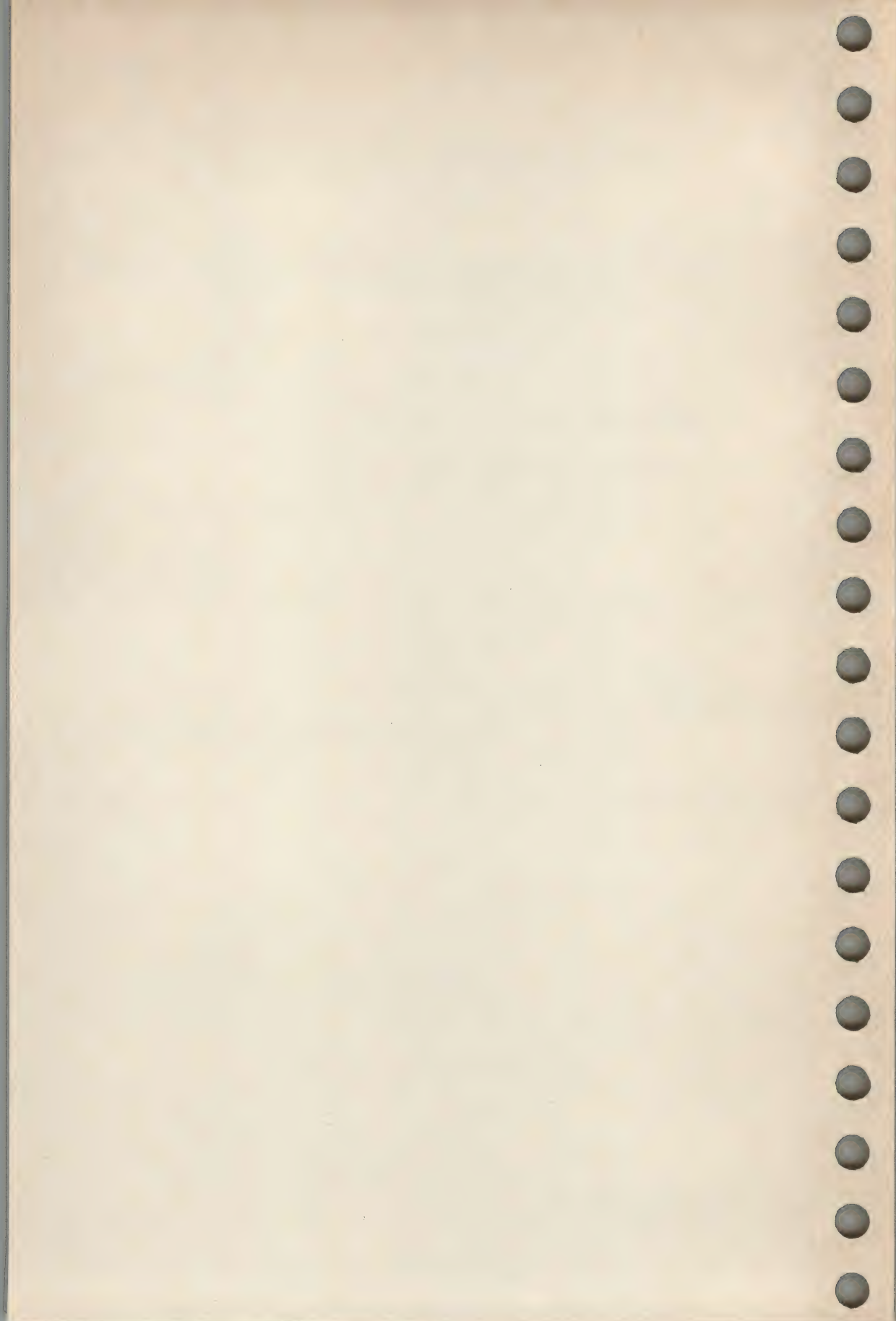
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

join — join two files on identical-valued field

## SYNOPSIS

join [ options ] file1 file2

## DESCRIPTION

The command *join* performs an "equality join" on the files *file1* and *file2*. If *file1* is — the standard input is used in its place.

A field must be specified for each file as the "join field" on which the files are compared. There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*. This format can be changed by using the —o option (see below).

The files *file1* and *file2* must be sorted in the collating sequence of *sort*, see *sort*(1), on the fields on which they are to be joined, normally the first in each line.

The default input field separators are blank, tab, or newline. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the options below use the argument *n*. This argument should be a 1 or a 2 referring to either *file1* or *file2*, respectively. The following options are recognised:

—an In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

—e *s* Replace empty output fields by string *s*.

—jn *m*

Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with 1.

—o *list*

Each output line comprises the fields specified in *list*, each element of which has the form *n.m* where *n* is a file number and *m* is a field number. The common field is not printed unless specifically requested.

—tc Use character *c* as a separator, for both input and output. Every appearance of *c* in a line is significant.

## SEE ALSO

awk(1), comm(1), sort(1), uniq(1).

## APPLICATION USAGE

Filenames that are numeric may cause conflict when the —o option is used immediately before listing filenames.



# JOIN(1)

*Utilities*

## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

In the third paragraph, the words "in increasing ASCII collating sequence" have been replaced by "in the collating sequence of *sort*, see *sort(1)*".

Spaces have been added to options in the **DESCRIPTION** where needed, to reflect the operation of System V Release 2.0.

## NAME

kill — send a signal to a process

## SYNOPSIS

kill [—signal] pid...

## DESCRIPTION

The command *kill* sends the specified *signal* to the specified processes (or process groups). If process number 0 is specified, all processes in the process group are signalled. Process numbers can be found by using *ps*, see *ps(1)*.

The argument *signal* must be specified as a numeric value; these values are implementation dependent. (See FUTURE DIRECTIONS below.)

If no signal is specified, *kill* sends SIGTERM (terminate). This will normally kill processes that do not catch or ignore the signal.

The specified process(es) must belong to the user unless the user is super-user.

## SEE ALSO

*ps(1)*, *kill(2)*, *signal(2)*, *signal(5)*.

## APPLICATION USAGE

Until symbolic naming is introduced, the following numeric values for the signals can be used:

- 1 SIGHUP
- 2 SIGINT
- 3 SIGQUIT
- 9 SIGKILL
- 15 SIGTERM

Refer to *signal(2)* for the definitions of these signals.

## FUTURE DIRECTIONS

The command *kill* will be changed to use symbolic names rather than numeric values of signals. The old form will continue to be accepted for some time.

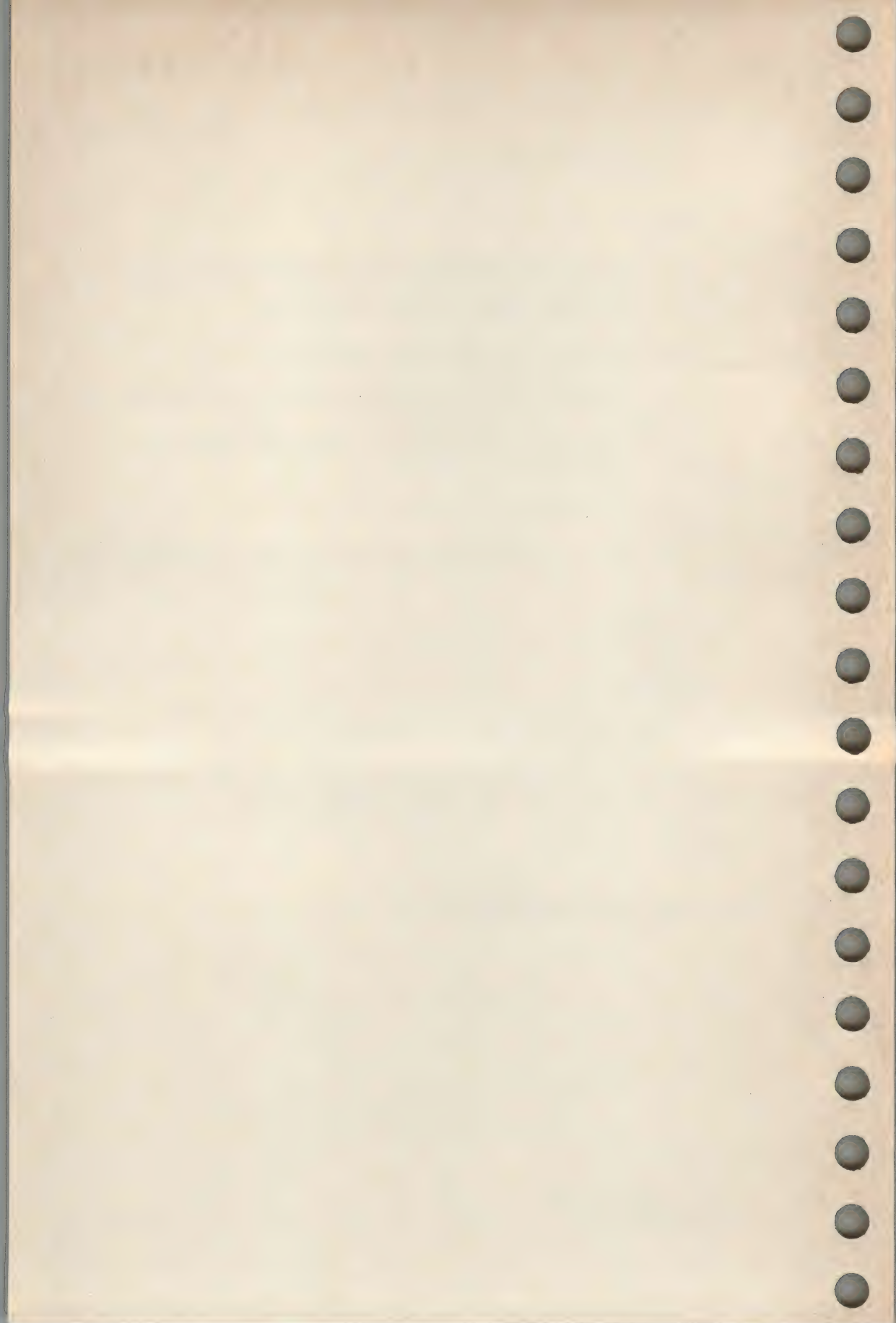
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

ld — link editor for object files

## SYNOPSIS

ld [options] file...

## DESCRIPTION

The *ld* command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and *ld* combines them, producing an object module that can either be executed or, if the *-r* option is specified, used as input for a subsequent *ld* run. The output of *ld* is left in *a.out*. By default this file is executable if no errors occurred during the load. If any input file *file* is not an object file, *ld* assumes it is an archive library.

If any argument is a library, it is searched at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table is searched to resolve external references which can be satisfied by library members. The ordering of library members is unimportant, unless there exist multiple library members defining the same external symbol.

The following options are recognised by *ld*:

*-e* *epsym*

Set the default entry point address for the output file to be that of the symbol *epsym*.

*-lxxx*

Search the library which has the abbreviation *xxx* (e.g., *-lm* to search the math library). A library is searched when its name is encountered, so the placement of a *-l* option is significant.

*-o* *outfile*

Produce an output object file by the name *outfile*. The name of the default object file is *a.out*.

*-r*

Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent *ld* run. The link editor will not complain about unresolved references, and the output file will not be made executable.

*-s*

Strip all symbolic information from the output object file.

*-u* *symname*

Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

UN

*-L dir*

Change the algorithm of searching for the library *xxx* to look in *dir* before looking in the default library directories. This option is effective only if it precedes the *-l* option on the command line.



MV UN **—V** Output a message giving information about the version of *ld* being used

#### FILES

a.out output file

#### SEE ALSO

ar(1), cc(1D), strip(1D), exit(2).

#### APPLICATION USAGE

When the link editor is called through *cc*, a startup routine is linked with the user's program. This routine calls *exit*(2) after execution of the main program. If the user calls the link editor directly, then the user must ensure that the program always calls *exit*(2) rather than falling through the end of the entry routine.

The symbols *etext*, *edata* and *end* as seen from a C program are reserved and are defined by the link editor. It is erroneous for a user program to redefine them.

The standard C library, containing routines for I/O, string and character manipulation, memory management, etc., can be searched using *—lc*. The math functions are contained in the optional math library which can be searched by *—lm*.

#### FUTURE DIRECTIONS

The SVID reserves the *—Y* option for future use. It will be used to specify a directory to be used instead of the standard list, when searching for libraries.

Users will also be able to specify, by means of the TMPDIR environment variable, the directory in which any temporary files are to be created.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

`lex` — generate programs for simple lexical analysis of text

## SYNOPSIS

`lex [—ctvn] [file...]`

## DESCRIPTION

The command `lex` generates programs to be used in lexical processing of character input and may be used as an interface to `yacc`.

The input *file(s)*, which contain `lex` source code, contain a table of regular expressions each with a corresponding action in the form of a C program fragment. Multiple input *files* are treated as a single file. When `lex` processes *file(s)*, this source is translated into a C program. Normally `lex` writes the program it generates to the file `lex.yy.c`. If the `—t` option is used, the resulting program is written instead to the standard output. When the program generated by `lex` is compiled and executed, it will read character input from the standard input and partition it into strings that match the given expressions. When an expression is matched, the input string that was matched is left in an external character array `yytext` and the expression's corresponding program fragment, or action, is executed. `Lex` also provides a count `yylen` of the number of characters matched. During pattern matching the set of patterns will be searched for a match in the order in which they appeared in the `lex` source and the single longest possible match will be chosen at any point in time. Among rules that match the same number of characters, the rule given first will be matched.

The program generated by `lex`, e.g., `lex.yy.c`, should be compiled and loaded with the `lex` library (using the `—l` option with `cc`).

MV UN

The option `—c` indicates C language actions and is the default. `—t` causes the program generated to be written instead to standard output; `—v` provides a one-line summary of statistics of the finite state machine generated, and `—n` will not print out the `—v` summary.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

<code>%p n</code>	number of positions is <i>n</i>
<code>%n n</code>	number of states is <i>n</i>
<code>%e n</code>	number of parse tree nodes is <i>n</i>
<code>%a n</code>	number of transitions is <i>n</i>
<code>%k n</code>	number of packed character classes is <i>n</i>
<code>%o n</code>	size of the output array is <i>n</i>

The use of one or more of the above automatically implies the `—v` option, unless the `—n` option is used.

The general format of `lex` source is:



```
{definitions}  
%%  
{rules}  
%%  
{user subroutines}
```

The definitions and the user subroutines may be omitted. The first %% is required to mark the beginning of the rules (regular expressions and actions); the second %% is required only if user subroutines follow.

In the definitions, any lines beginning with whitespace, and any lines included between lines containing only %{ and %}, are assumed to contain only C text and are copied unchanged into the external definition area of the *lex.yy.c* file. Such lines may also appear between the first %% delimiter and the first *lex* rule but, in this case, they are copied unchanged into the internal definition area of the program *yylex()* generated by *lex*. All text after the second %% delimiter is copied unchanged to *lex.yy.c*.

#### Definitions

Definitions must appear before the first %% delimiter. Any line in this section not contained between %{ and %} lines and beginning in column 1 is assumed to define a *lex* substitution string. The format of these lines is

```
name substitute
```

The *name* must begin with a letter and be followed by at least one blank or tab. The *substitute* will replace the string {*name*} when it is used in a rule. The braces do not imply parentheses; only string substitution is done.

#### Rules

The rules in *lex* source files are a table in which the left column contains regular expressions and the right column contains actions and program fragments to be executed when the expressions are recognised.

```
re whitespace action  
re whitespace action  
...
```

Because the regular expression, *re*, portion of a rule is terminated by the first blank or tab, any blank or tab used within a regular expression must be quoted (its special meaning escaped). That is, it must appear within double quotes or square brackets or must be preceded by a backslash character.

The program fragment which is the action associated with a particular *re* may extend across several lines if it is enclosed in braces:

```
re whitespace { program statement  
                program statement }
```

#### Regular Expressions

The *lex* command supports extended regular expression syntax, with some additional expressions. Below is a summary of the additions.

1. Characters surrounded by double quotes, "*...*", have no special meaning.
2. The notation *<s>r* matches the regular expression *r* only when the program is in the start condition (state), *s*.
3. The notation *r/x* matches the regular expression *r* only if it is followed by the regular expression *x*. (Note this is *r* in the context of *x* and only *r* is matched.)
4. The notation {*S*} matches the substitution of *S* from the *definitions* section.

#### Actions

The default action when a string in the input to a *lex.yy.c* program is not matched by any expression is to copy the string to the output. Because the default behaviour of a program generated by *lex* is to read the input and copy it to the output, a minimal *lex* source program that has just `%%` will generate a C program that simply copies the input to the output unchanged. A null C statement, the statement `;`, may be specified as an action in a rule. Any string in the *lex.yy.c* input that matches the pattern portion of such a rule will be effectively ignored or skipped.

Three special actions are available; `|`, *REJECT*, and *ECHO*. The action `|` means that the action for the next rule is the action for this rule. *ECHO* prints the string *yytext* on the output. Normally only a single expression is matched by a given string in the input. *REJECT* means "continue to the next expression that matches the current input" and causes whatever rule was second choice after the current rule to be executed for the same input. Thus, it allows multiple rules to be matched and executed for one input string or overlapping input strings. For example, given the expressions *xyz* and *yz* and the input *xyz*, normally only one pattern, *xyz*, would match and the next attempted match would start at *z*. If the last action in the *xyz* rule is *REJECT*, both this rule and the *yz* rule would be executed.

The *lex* command provides several routines that can be used in the *lex* source program: *yymore()*; *yyless(n)*; *input()*; *output(c)*, and *unput(c)*.

The function *yymore()* may be called to indicate that the next input string recognised is to be concatenated onto the end of the current string in *yytext* rather than overwriting it in *yytext*. *Yyless(n)* returns to the input some of the characters matched by the currently successful expression. The argument *n* indicates the number of initial characters in *yytext* to be retained; the remaining trailing characters in *yytext* are returned to the input.

*input()*

returns the next character from the input. *Input* returns 0 on end of file.

*unput(c)*

pushes the character *c* back onto the input stream to be read later by *input()*.

*output(c)*

writes the character *c* on the output.



To perform custom processing when the end of input is reached, users may supply their own *yywrap()* function. *Yywrap()* is called whenever *lex.yy.c* reaches an end-of-file. If *yywrap()* returns a one, *lex.yy.c* continues with the normal wrap-up on end of input. The default *yywrap()* always returns a one. If the user wants *lex.yy.c* to continue processing with another source of input, then a *yywrap()* must be supplied that arranges for the new input and returns a zero. These routines may be redefined by the user.

The external names generated by *lex* all begin with the prefix *yy* or *YY*.

The program generated by *lex* is named *yylex()*; if the user does not supply a main routine, the default *main()* routine calls *yylex()*. If the user supplies a *main()* routine, it should call *yylex()*.

#### EXAMPLE

```
D      [0-9]
%%
if      printf("IF statement \n");
[a-z]+  printf("tag, value %s \n",yytext);
0{D}+   printf("octal number %s \n",yytext);
{D}+    printf("decimal number %s \n",yytext);
"++"    printf("unary op \n");
"++"    printf("binary op \n");
"/*"    {
        loop:
            while (input() != '*');
            switch (input()) {
            case '/':
                break;
            case '*':
                unput('*');
            default:
                goto loop;
            }
        }
```

#### FILES

*lex.yy.c*                      output file generated by *lex*.

#### SEE ALSO

*cc*(1D), *yacc*(1D).

CHANGE HISTORY

First released in Issue 2.

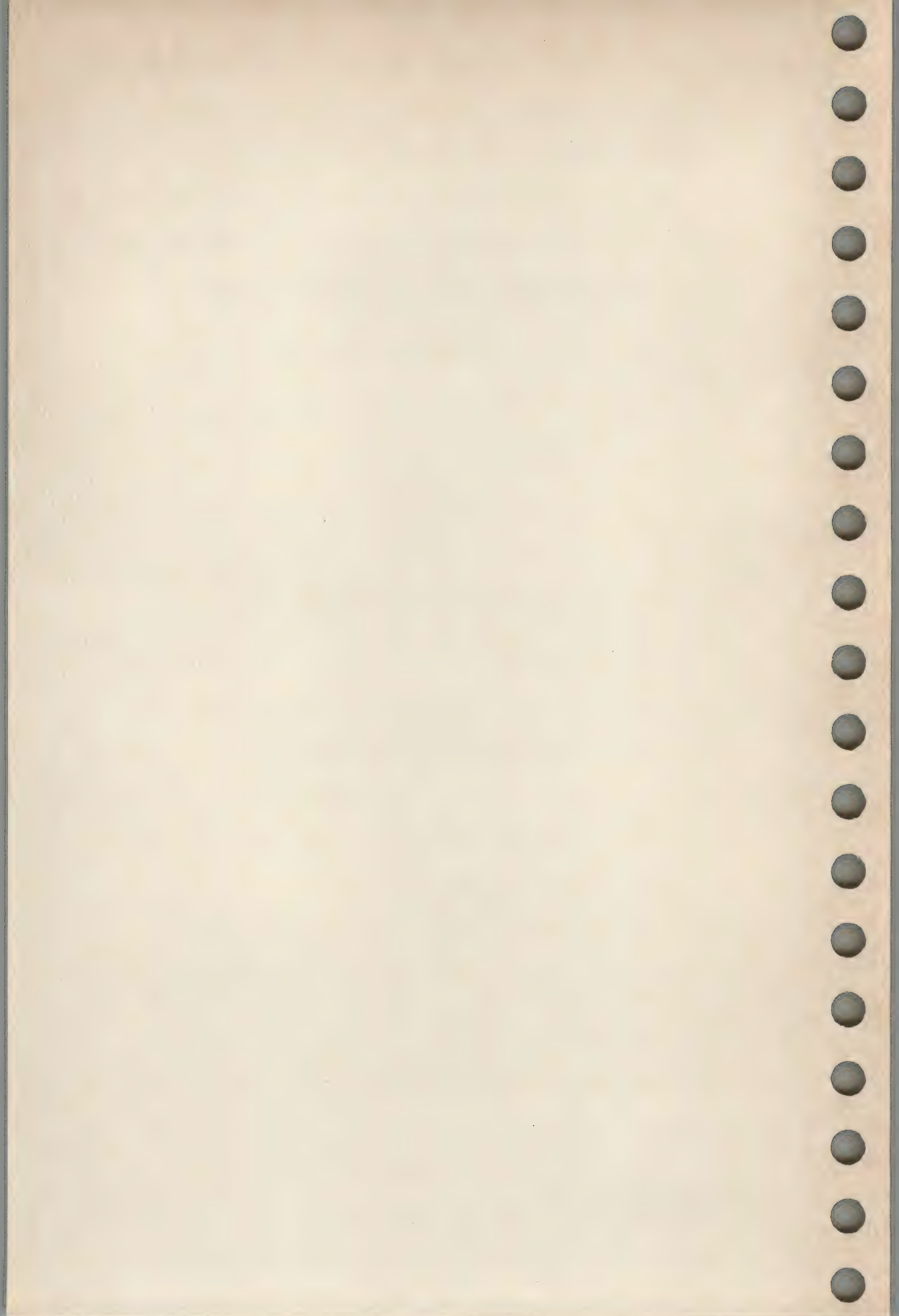
Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

In the second paragraph an erroneous reference to the variable *yylex* has been replaced by *yytext*.

A sentence indicating that *lex* provides a count *yylen*g has been added.





## NAME

*line* — read one line

## SYNOPSIS

*line*

## DESCRIPTION

The command *line* copies one line (up to and including a newline) from the standard input and writes it on the standard output. It always prints at least a newline.

## EXIT STATUS

Exit status is:

- 0 normal exit
- 1 EOF on input

## SEE ALSO

*sh*(1).

## APPLICATION USAGE

*Line* is often used within command scripts to read from the user's terminal.

## CHANGE HISTORY

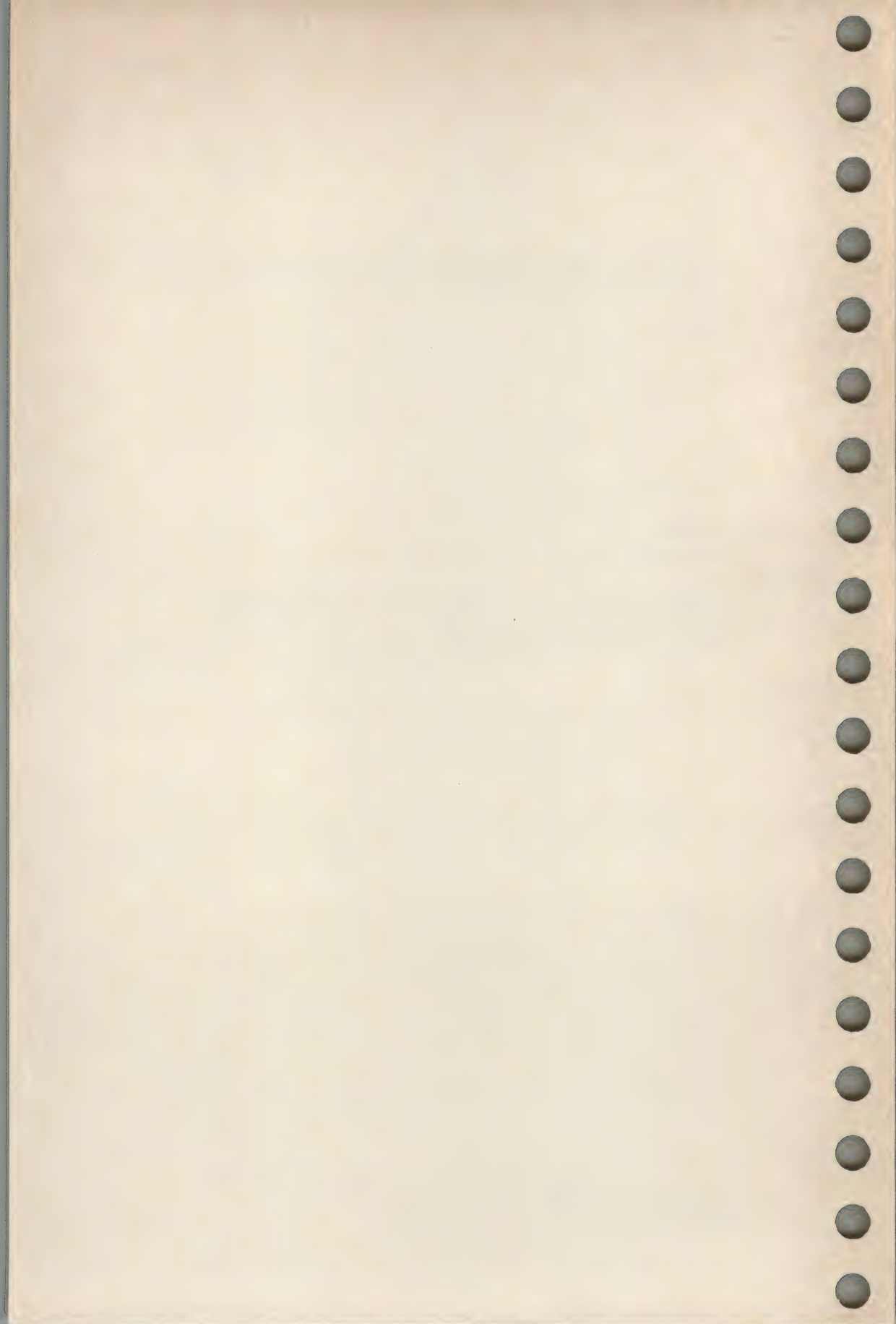
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

Inclusion of the newline character has been added.





## NAME

lint — a C program checker

## SYNOPSIS

lint [ options ] file ...

## DESCRIPTION

The command *lint* attempts to detect features of the C program *file* that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

The options are described below.

Arguments whose names end with *.c* are taken to be C source files.

Arguments whose names end with *.ln* are taken to be the result of an earlier invocation of *lint* with either the *—c* or the *—o* option used. The *.ln* files are analogous to *.o* (object) files that are produced by the *cc* command when given a *.c* file as input.

Files with other suffixes are warned about and ignored.

The command *lint* will take all the *.c*, *.ln*, and *llib-lxxx* (specified by *—lxxx*) files and process them in their command line order. By default, *lint* appends the standard C lint library to the end of the list of files. However, if the *—p* option is used, the portable C lint library (*llib-port.ln*) is appended instead. When the *—c* option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the *—c* option is used, the *.ln* files and the lint libraries are ignored.

Any number of *lint* options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

- PI *—a* Suppress complaints about assignments of long values to variables that are not long.
- PI *—b* Suppress complaints about *break* statements that cannot be reached (Programs produced by *lex* or *yacc* will often result in many such complaints.)
- PI *—h* Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.
- PI *—u* Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program.)
- PI *—v* Suppress complaints about unused arguments in functions.
- PI *—x* Do not report variables referred to by external declarations but never used.



The following arguments alter *lint*'s behaviour:

- lxxx  
Include additional lint library xxx (e.g., —lm for the math library).
- n Do not check compatibility against either the standard or the portable *lint* library.
- p Attempt to check portability.
- c Cause *lint* to produce a .ln file for every .c file on the command line. These .ln files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.
- o lib  
Cause *lint* to create a lint library with the name *lib*. The —c option nullifies any use of the —o option. The lint library produced is the input that is given to *lint*'s second pass. The —o option simply causes this file to be saved in the named lint library. To produce the lint library without extraneous messages, use of the —x option is suggested. The —v option is useful if the source file(s) for the lint library are just external interfaces. These option settings are also available through the use of "lint comments" (see below).

The —D, —U, and —I options of the C preprocessor, see *cpp*(1D), are also recognised as separate arguments.

The —g and —O options of *cc* are also recognised as separate arguments. These options are ignored but, by recognising these options, *lint*'s behaviour is closer to that of the *cc* command. Other options are warned about and ignored. The preprocessor symbol *lint* is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol *lint* should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behaviour of *lint*:

/\*NOTREACHED\*/

at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions like *exit*.)

/\*VARARGSn\*/

suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/\*ARGSUSED\*/

turns on the —v option for the next function.

/\*LINTLIBRARY\*/

at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the —v and —x options.

The command *lint* produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the `—c` option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behaviour of the `—c` and the `—o` options allows for incremental use of *lint* on a set of C source files. Generally, *lint* is invoked once for each source file with the `—c` option. Each of these invocations produces a *.ln* file which corresponds to the *.c* file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the `—c` option), listing all the *.ln* files with the needed `—lxxx` options. This will print all the inter-file inconsistencies.

This scheme works well with *make*; it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were checked by *lint*.

#### SEE ALSO

`cc(1D)`, `cpp(1D)`, `make(1D)`.

#### APPLICATION USAGE

On some implementations, the behaviour of the `—a`, `—b`, `—h`, `—u`, `—v`, and `—x` options is reversed in that their presence enables the messages.

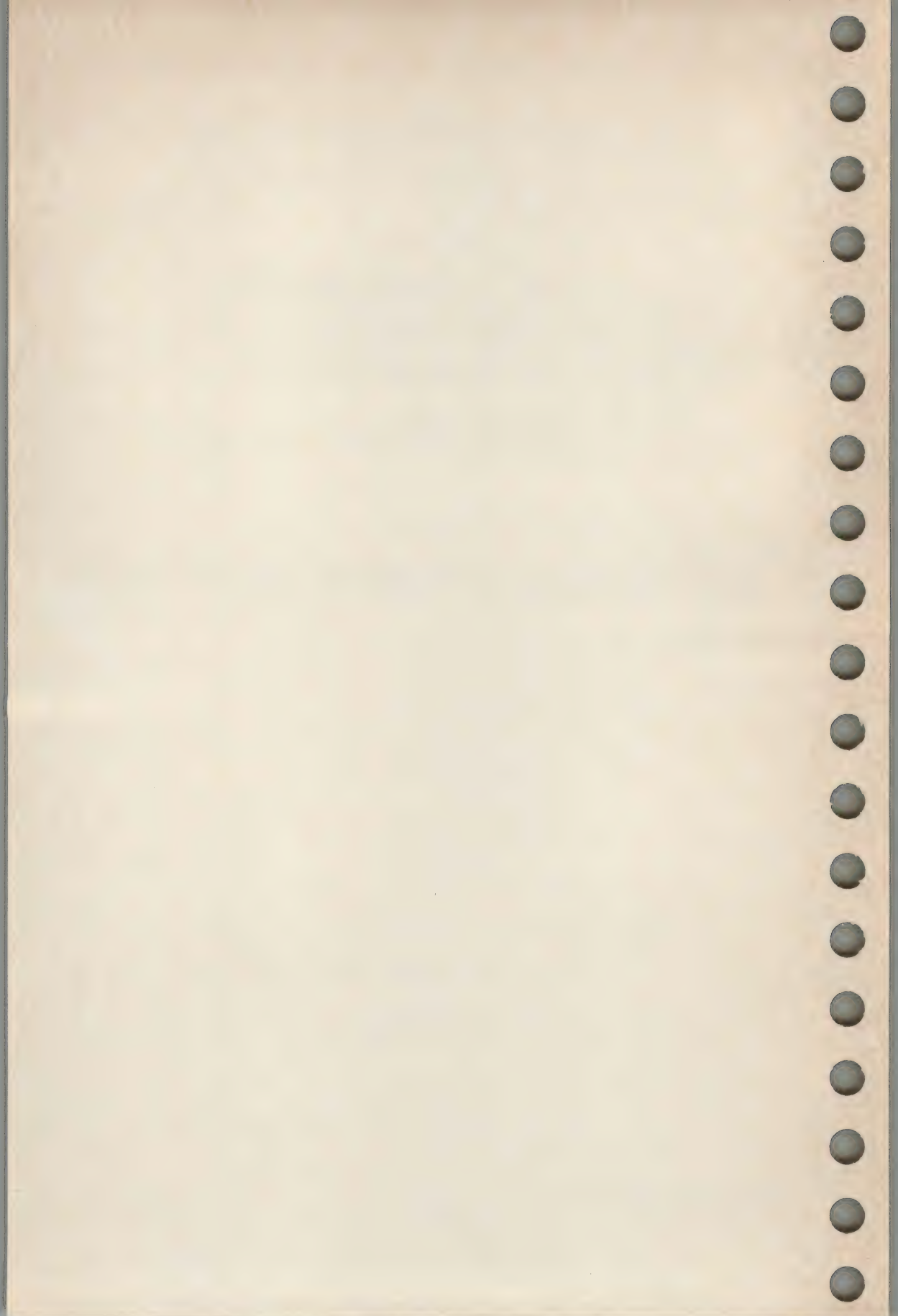
#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID.





NAME

logname — get login name

SYNOPSIS

logname

DESCRIPTION

The command *logname* writes the user's login name on the standard output.

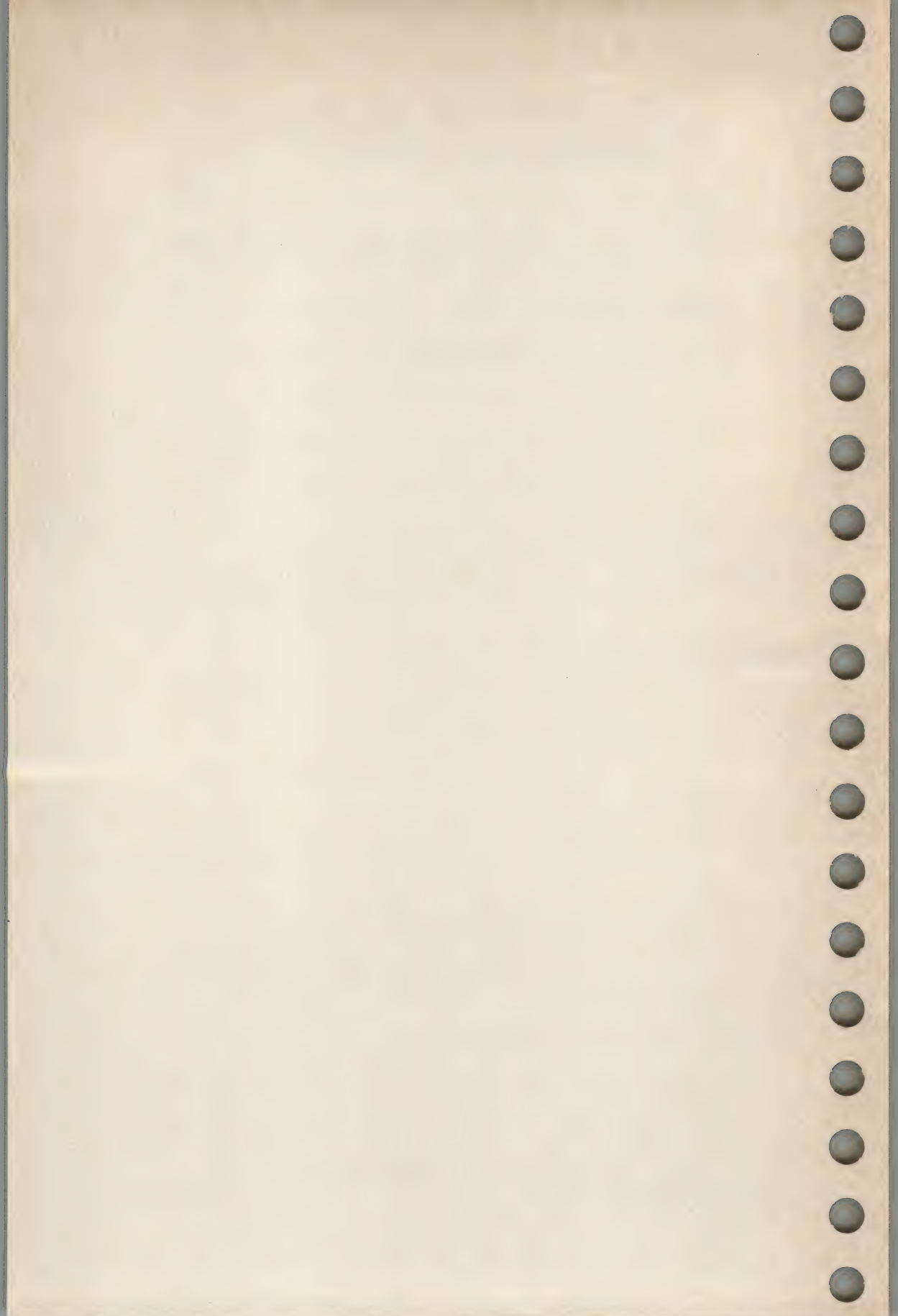
CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`lorder` — find ordering relation for an object library

## SYNOPSIS

`lorder file...`

## DESCRIPTION

The input is one or more object or library archive *files*, see *ar(1)*. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort(1D)* to find an ordering of a library suitable for one-pass access by the link editor, *ld*.

## EXAMPLE

The following example builds a new library from existing `.o` files.

```
ar -cr library `lorder *.o | tsort`
```

## SEE ALSO

*ar(1)*, *ld(1D)*, *tsort(1D)*.

## APPLICATION USAGE

Note that *ld* is capable of multiple passes over an archive in the *ar(1)* format and does not require that *lorder* be used when building an archive. The usage of the *lorder* command may, however, allow for a slightly more efficient access of the archive during the link edit process.

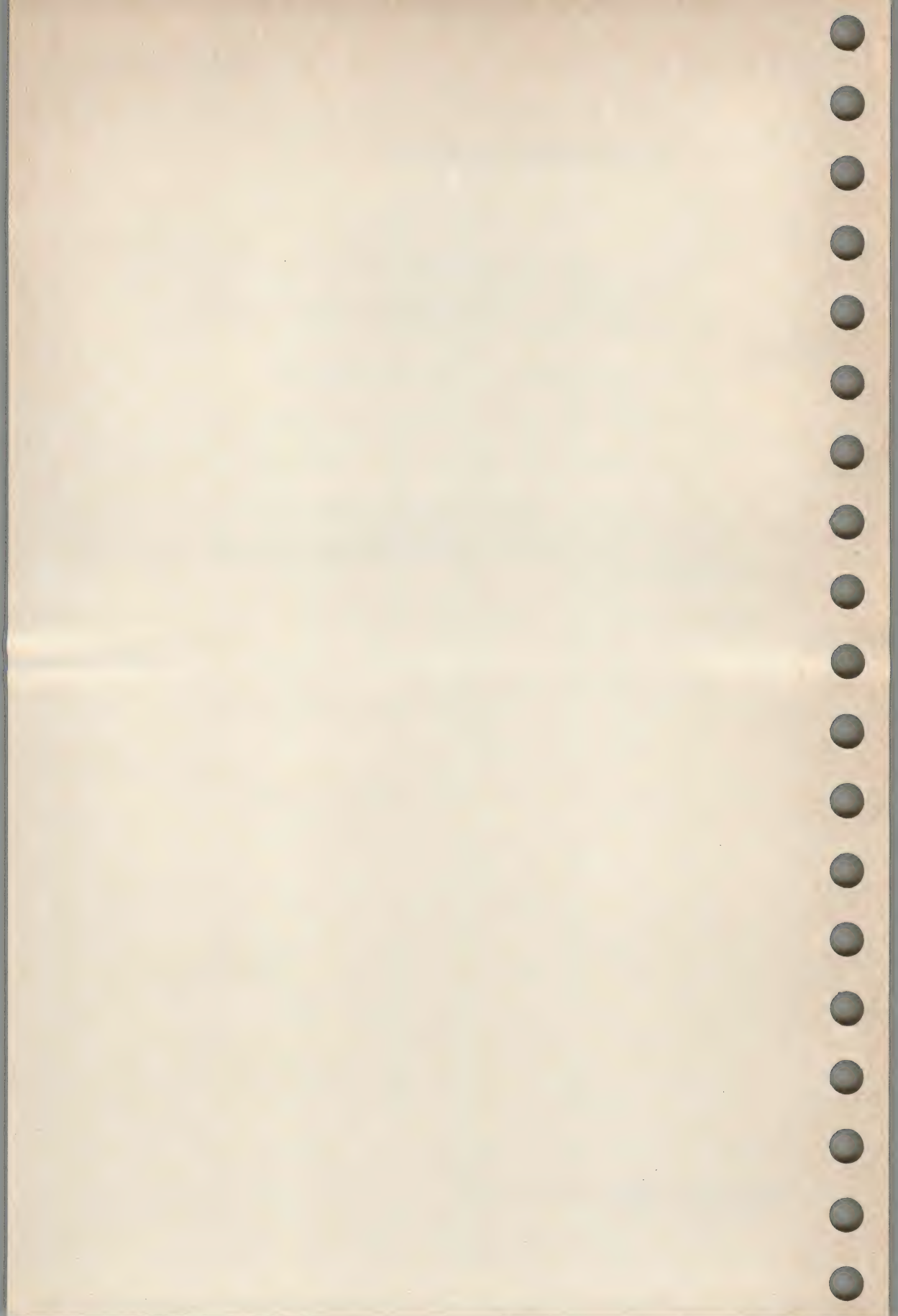
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`lp`, `cancel` — send/cancel line printer requests

## SYNOPSIS

`lp [-c] [-ddest] [-m] [-nnumber] [-ooption] [-s] [-ttitle] [-w]  
[file...]`

`cancel [id...] [printer...]`

UN

## DESCRIPTION

`lp` The command `lp` arranges for the named *file* or files and associated information (collectively called a *request*) to be printed by a line printer. If no file names are mentioned, the standard input is assumed. The file name — stands for the standard input and may be supplied on the command line in conjunction with named files. The order in which files appear is the same order in which they will be printed.

OF `lp` associates a unique *id* with each request and prints it on the standard output. This *id* can be used later to cancel, see `cancel`, or find the status, see `lpstat(1)`, of the request.

The following options to `lp` may appear in any order and may be intermixed with file names:

—c Make copies of the files to be printed immediately when `lp` is invoked. Normally, files will not be copied, but will be linked whenever possible. If the —c option is not given, then the user should be careful not to remove any of the files before the request has been printed in its entirety. It should also be noted that in the absence of the —c option, any changes made to the named files after the request is made but before it is printed will be reflected in the printed output. On some systems, —c may be on by default.

—ddest

Choose *dest* as the printer or class of printers that is to do the printing. If *dest* is a printer, then the request will be printed only on that specific printer. If *dest* is a class of printers, then the request will be printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, etc.), requests for specific destinations may not be accepted, see `lpstat(1)`. By default, *dest* is taken from the environment variable `LPDEST` (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems, see `lpstat(1)`.

UN —m Send mail, see `mail(1)`, after the files have been printed. By default, no mail is sent upon normal completion of the print request.

—nnumber

Print *number* copies (default of 1) of the output.

UN —ooption

Specify printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the —o option more than once.



PI **—s** Suppress messages from *lp* such as "request id is ...".

UN **—ttitle**  
Print *title* on the banner page of the output.

UN **—w** Write a message on the user's terminal after the files have been printed. If the user is not logged in, then mail will be sent instead.

**cancel**

The command *cancel* cancels line printer requests that were made by the *lp* command. The command line arguments may be either request *ids* (as returned by *lp*) or printer names (for a complete list, use *lpstat(1)*). Specifying a request *id* cancels the associated request even if it is currently printing. Specifying a printer cancels the request which is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

**SEE ALSO**

*lpstat(1)*, *mail(1)*.

**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID with the following change:

The sentence "On some systems, **—c** may be on by default." has been added to the description of the **—c** option.

## NAME

lpstat — print line printer status information

## SYNOPSIS

**lpstat [ options ] [ id . . . ]**

UN OF PI

## DESCRIPTION

The command *lpstat* prints information about the current status of the line printer system.

If no *options* are given, then *lpstat* prints the status of all requests made to *lp(1)* by the user. Any arguments that are not *options* are assumed to be request *ids* as returned by *lp*, see *lp(1)*. The command *lpstat* prints the status of such requests. The *options* may appear in any order and may be repeated and intermixed with other arguments. Some of the options below may be followed by an optional *list* that can be in one of two forms: a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

lpstat —u"user1, user2, user3"

The omission of a *list* following such options causes all information relevant to the option to be printed, for example:

lpstat

prints the status of all output requests.

—a[ *list* ]

Print acceptance status of destinations for output requests. *List* is a list of intermixed printer names and class names.

—c[ *list* ]

Print class names and their members. *List* is a list of class names.

—d   Print the system default destination for output requests.

—o[ *list* ]

Print the status of output requests. *List* is a list of intermixed printer names, class names, and request *ids*.

—p[ *list* ]

Print the status of printers. *List* is a list of printer names.

—r   Print the status of the line printer request scheduler.

—s   Print a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, and a list of printers and their associated devices.

—t   Print all status information.

—u[ *list* ]

Print status of output requests for users. *List* is a list of login names.



## LPSTAT(1)

Utilities

—v[*list*]

Print the names of printers and the path names of the devices associated with them. *List* is a list of printer names.

### SEE ALSO

lp(1).

### CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

ls — list contents of directory

## SYNOPSIS

ls [ options ] [ file ... ]

## DESCRIPTION

For each *file*, if it is a directory, *ls* lists the contents of the directory; if it is a file, *ls* repeats its name and gives any other information requested. The output is sorted alphabetically by default. When no *files* are specified, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but files appear before directories and their contents.

There are three major listing formats. The default format is to list one entry per line; the options *—C* and *—x* enable multi-column formats, and the *—m* option enables stream output format in which files are listed across the page, separated by commas.

In order to determine output formats for the *—C*, *—x*, and *—m* options, *ls* uses the environment variable COLUMNS to determine the number of character positions available on one output line. If this variable is not set, *ls* may optionally attempt to determine this information from a system-dependent database, keyed on the value of the TERM environment variable. If this information cannot be obtained, 80 columns is assumed.

There are a large number of options:

- OF *—C* Multi-column output with entries sorted down the columns.
- OF *—F* Put a slash (/) after each filename if that file is a directory and put an asterisk (\*) after each filename if that file is executable. For other file types, other symbols may be printed.
- OF *—R* Recursively list subdirectories encountered.
- a* List all entries; usually entries whose names begin with a period (.) are not listed.
- c* Use time of last modification of the file status information (file created, mode changed, etc.) for sorting (*—t*) or printing (*—l*).
- d* If an argument is a directory, list only its name (not its contents); often used with *—l* to get the status of a directory.
- f* Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off *—l*, *—t*, *—s*, and *—r*, and turns on *—a*; the order is the order in which entries appear in the directory.
- OF *—g* The same as *—l*, except that the owner is not printed.
- i* For each *file*, print its unique identification number in the first column of the report.



- OF —l List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each *file* (see below). If the file is a special file, the size field will instead contain the major and minor device numbers rather than a size.
- OF —m Stream output format; files are listed across the page, separated by commas.
- OF UN —n The same as —l, except that the owner's *UID* and group's *GID* numbers are printed, rather than the associated character strings.
- OF —o The same as —l, except that the group is not printed.
- p Put a slash (/) after each filename if that file is a directory.
- q Force non-printing characters (in file names) to be displayed as the character ?.
- r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.
- OF —s Give total number of 512-byte units consumed by each *file*.
- t Sort by time modified (latest first) instead of by name.
- u Use time of last access instead of last modification for sorting (with the —t option) or printing (with the —l option).
- OF UN —x Multi-column output with entries sorted across rather than down the page.

The mode printed under the —l option consists of 10 characters that are interpreted as described below.

The first character is:

- d* if the entry is a directory;
- b* if the entry is a block special file;
- c* if the entry is a character special file;
- UN *p* if the entry is a *fifo* (named pipe) special file;
- if the entry is an ordinary file.

On some implementations, other characters may be printed for implementation-dependent file types.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the file, and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, execute permission is interpreted to mean permission to search the directory for a specified file.

The permissions are indicated as follows:

- r* if the file is readable
- w* if the file is writable
- x* if the file is executable (also see below)
- if the indicated permission is not granted

The group-execute permission character is given as *s* if the file has set-group-ID mode; likewise, the user-execute permission character is given as *s* if the file has set-user-ID mode. These are given as *S* (capitalised) if the corresponding execute permission is not set.

When the sizes of the files in a directory are listed, a total count of 512-byte units consumed by the directory is printed.

#### FILES

/etc/passwd	to get user IDs for <i>ls -l</i> and <i>ls -o</i> .
/etc/group	to get group IDs for <i>ls -l</i> and <i>ls -g</i> .

#### SEE ALSO

chmod(1), find(1).

#### APPLICATION USAGE

Currently not all systems report in terms of 512-byte units. This situation may change in the future.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

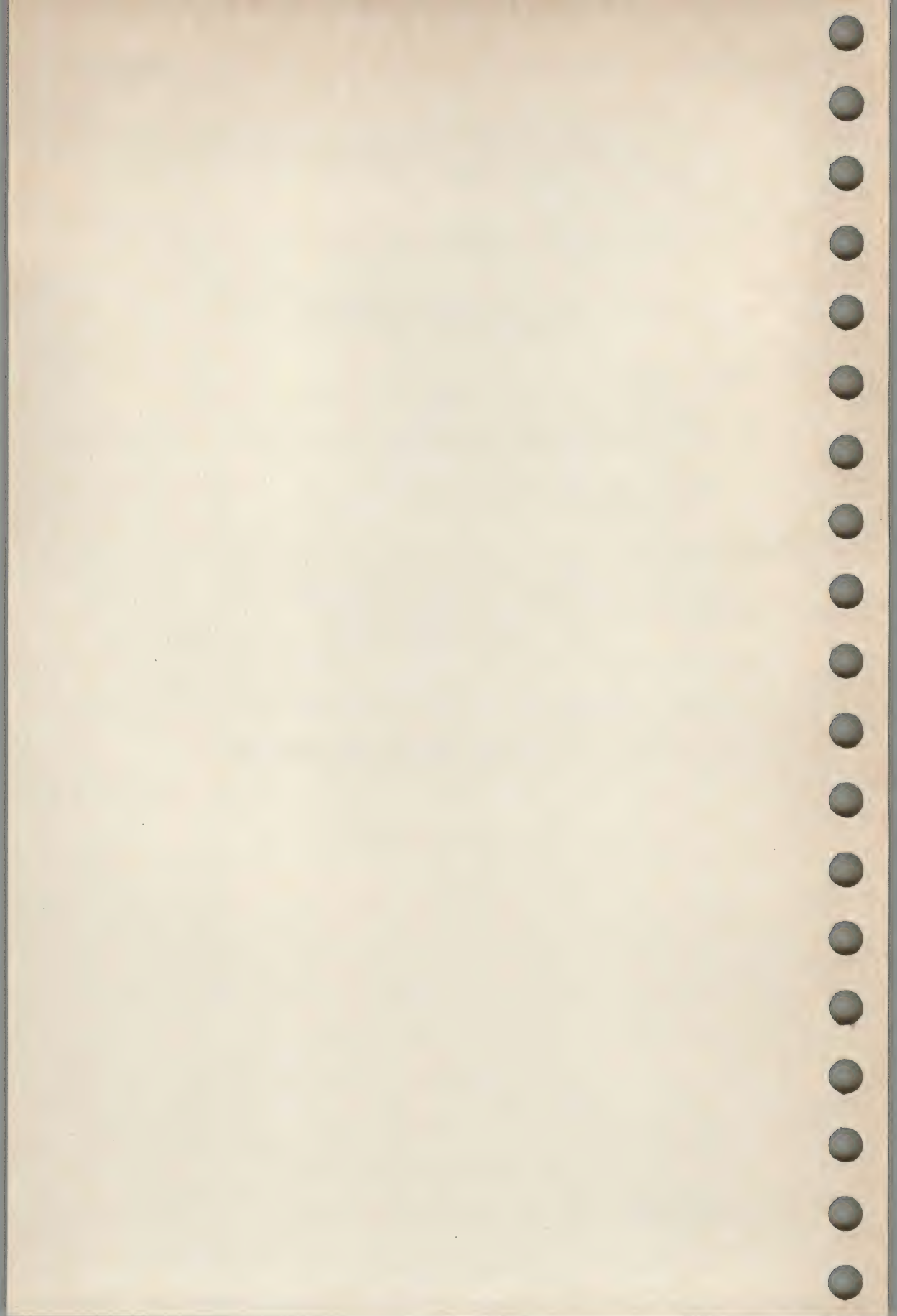
The paragraph reading "When the sizes of the files in a directory are listed, a total count of 512-byte units consumed by the directory is printed." has been added.

In the description of the *-F* option, the paragraph referring to other symbols has been added.

The paragraph referring to other characters being printed for implementation-dependent file types has been added.

The FUTURE DIRECTION on mandatory file locking has been omitted.





## NAME

m4 — macro processor

## SYNOPSIS

m4 [options] [file...]

## DESCRIPTION

The command *m4* is a macro processor intended as a front end for C and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is —, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

—s Enable line sync output for the C preprocessor (i.e., *#line* directives).

This option must appear before any file names and before the following options.

—Dname[=val]

Defines *name* to *val* or to null in *val*'s absence.

—Uname

undefines *name*.

Macro calls have the form:

name(arg1, arg2, ..., argn)

The ( must immediately follow the name of the macro. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore \_, where the first character is not a digit.

Leading unquoted blanks, tabs and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognised, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null.

Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which appear within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

The command *m4* makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.



**define**

the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of  $\$n$  in the replacement text, where  $n$  is a digit, is replaced by the  $n$ th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string;  $\$ \#$  is replaced by the number of arguments;  $\$ *$  is replaced by a list of all the arguments separated by commas, and  $\$ @$  is like  $\$ *$ , but each argument is quoted (with the current quotes).

**undefine**

removes the definition of the macro named in its argument.

**defn** returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

**pushdef**

like *define*, but saves any previous definition.

**popdef**

removes current definition of its argument(s), exposing the previous one, if any.

**ifdef** if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null.

**shift** returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

**changequote**

changes quote symbols to the first and second arguments. The symbols may be up to five characters long. The command *changequote* without arguments restores the original values (i.e., `').

**changecom**

changes left and right comment markers from the default  $\#$  and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

**divert**

The command *m4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

**undivert**

causes immediate output of text from diversions named as arguments, or all diversions if no arguments are present. Text may be undiverted into another diversion. Undiverting discards the diverted text.

**divnum**

returns the value of the current output stream.

**dnl** reads and discards characters up to and including the next newline.

**ifelse**

has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string or, if it is not present, null.

**incr** returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

**decr** returns the value of its argument decremented by 1.

**eval** evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, -, \*, /, %, ^ (exponentiation), bitwise &, |, ^, and ~; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

**len** returns the number of characters in its argument.

**index**

returns the position in its first argument where the second argument begins (zero origin), or -1 if the second argument does not occur.

**substr**

returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

**translit**

transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

**include**

returns the contents of the file named in the argument.

**sinclude**

is identical to *include*, except that it says nothing if the file is inaccessible.

**syscmd**

executes the system command given in the first argument. No value is returned.

**sysval**

is the return code from the last call to *syscmd*.

**maketemp**

fills in a string of XXXXX in its argument with the current process ID.



## **m4exit**

causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.

## **m4wrap**

argument 1 will be pushed back at final EOF; example: *m4wrap(cleanup())*

## **errprint**

prints its argument on the diagnostic output file.

## **dumpdef**

prints current names and definitions for the named items, or for all if no arguments are given.

## **traceon**

with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.

## **traceoff**

turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

## SEE ALSO

*cc*(1D), *cpp*(1D).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

A reference to *Ratfor* has been deleted.

## NAME

mail — send or read mail

## SYNOPSIS

mail [ **-e** *pqr* ] [ **-f** *file* ]

mail [ **-t** ] *name*...

## DESCRIPTION

## Reading Mail

The command *mail* without arguments prints a user's mail, message-by-message. Mail is stored in the user's individual *mailfile*. For each message, the user is given a prompt and a line is read from the standard input to determine the disposition of the message:

<newline>

Go on to next message.

+ Same as <newline>.

d Delete message and go on to next message.

p Print message again.

— Go back to previous message.

s [ *file* ]

Save message in the named *file* (*mbox* is default).

PI w [ *file* ]

Save message, (on some implementations, without its header), in the named *file* (*mbox* is default).

PI m [ *name* ... ]

Mail the message to the named users; the default is the user.

PI q Store undeleted mail and stop.

<EOF>

Same as q.

x Put all mail back in the *mailfile* unchanged and stop.

!*command*

Escape to the command interpreter to execute *command*.

UN • Print a command summary

The optional arguments alter the printing of the mail:

—e Causes mail not to be printed. An exit value is returned; see EXIT STATUS.

UN —p Causes all mail to be printed without prompting for disposition.

UN —q Causes *mail* to terminate after interrupts. Normally an interrupt only causes the termination of the message being printed.



UN **—r** Causes messages to be printed in first-in, first-out order.

**—f file**

Causes *mail* to use *file* (e.g., *mbox*) instead of the default *mailfile*.

### Sending Mail

When *names* (user login names) are given, *mail* takes the standard input up to an end-of-file (or up to a line consisting of just a *.*) and adds it to each user's *mailfile*. The message is preceded by the sender's name and a postmark. Lines in the message that begin with the sequence "*From* " are preceded with a *>*.

UN The **—t** option causes the message to be preceded by all users the *mail* is sent to

If a user being sent mail is not recognised, or if *mail* is interrupted during input, the message is saved in the file *dead.letter* to allow editing and resending. Note that this is regarded as a temporary file in that it is recreated every time it is needed, erasing the previous contents of *dead.letter*.

PI It may also be possible to send mail to remote systems using system specific naming conventions.

PI There are implementation-specific mechanisms which can be used to cause all mail sent to the user to be forwarded to one or more other destinations.

### FILES

/etc/passwd	to identify sender and locate persons
\$HOME/mbox	saved mail
dead.letter	unmailable text

### EXIT STATUS

If *mail* is invoked with the **—e** option, the following exit values are returned:

0	the user has mail
1	the user has no mail

### APPLICATION USAGE

The forwarding facility is especially useful to forward all of a person's mail to one machine in a multiple machine environment.

Delivery of messages to remote systems requires the existence of communications paths to such systems. These may not exist.

The location of stored mail on exiting from *mail* using the *q* command differs between implementations and may be either the user's *mailfile* or the user's *mbox*.

In the description of reading mail, the phrase "go on to next message" may or may not imply the printing of the next message.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The sentence "Mail is stored in the user's individual *mailfile*." has been added to the first paragraph of the description.

References to *rmail* have been removed as this command is implementation-specific.

The words "in last-in first-out order" have been deleted from the first paragraph of the description of reading *mail*, and the words "given a prompt" have been changed from "prompted with a ?".

The words "on some implementations" have been added to the description of the *w* command.

The words "names are user login names" have been deleted from the description of the *m* command.

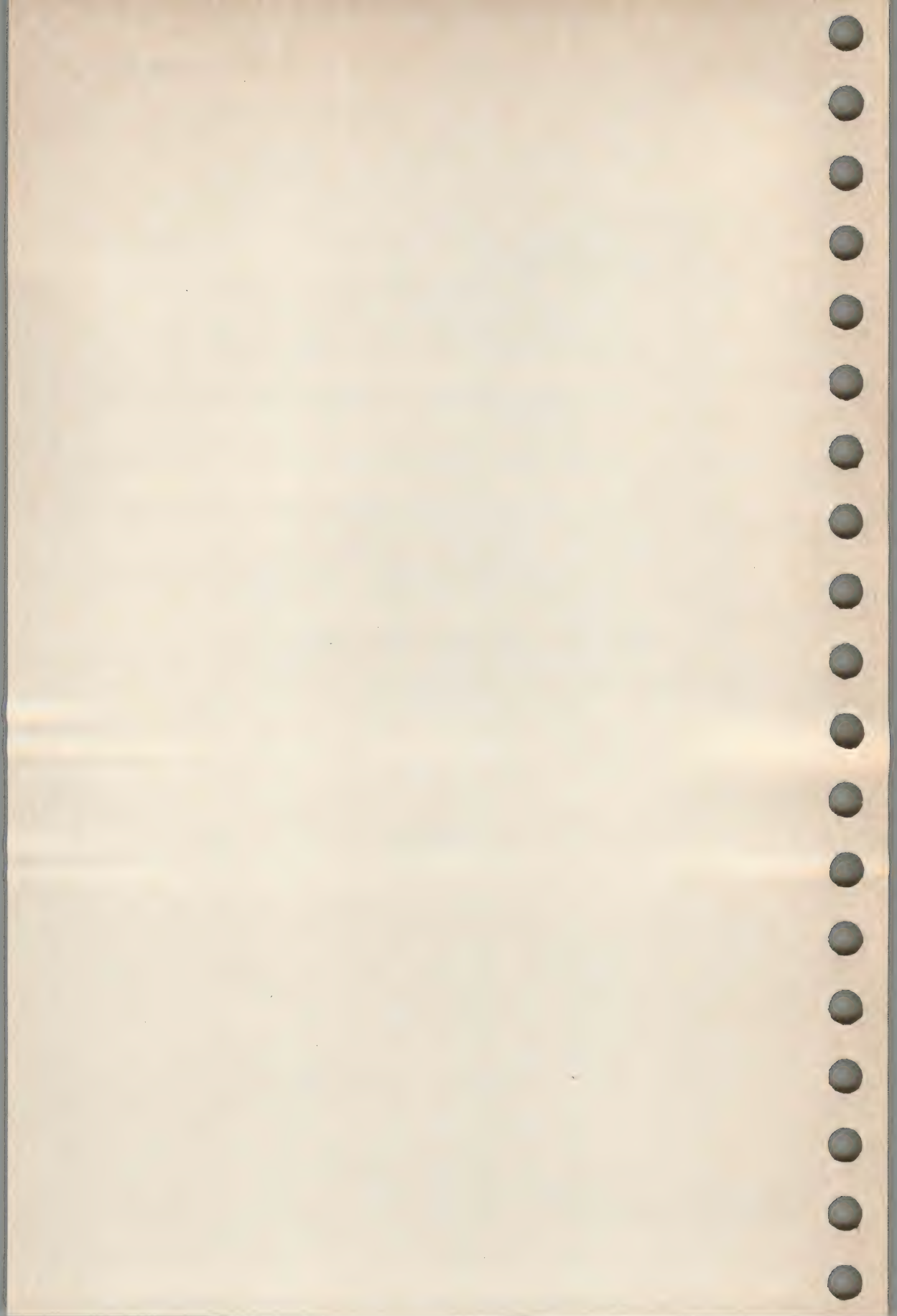
The description of the *q* command has been changed from "Put undeleted mail back in the *mailfile* and stop" to "Store undeleted mail and stop".

In the description of errors which occur while mail is being sent, the words "the file *dead.letter* will be saved" have been changed to "the message is saved in the file *dead.letter*" to improve clarity.

The description of sending mail to remote systems has been simplified.

The description of forwarding mail has been changed to reflect different implementations of *mail*.





## NAME

mailx — interactive message processing system (OPTIONAL)

## SYNOPSIS

**mailx [ options ] [ name ... ]**

UN

## DESCRIPTION

The command *mailx* provides a comfortable, flexible environment for sending and receiving messages electronically. When reading mail, *mailx* provides commands to facilitate saving, deleting, and responding to messages. When sending mail, *mailx* allows editing, reviewing and other modification of the message as it is entered.

Incoming mail is stored in a standard file for each user, called the system *mailbox* for that user. When *mailx* is called to read messages, the *mailbox* is the default place to find them. As messages are read, they are marked to be moved to a secondary file for storage unless specific action is taken, so that the messages need not be seen again. This secondary file is called the *mbox* and is normally located in the user's *HOME* directory (see *MBOX* in *Environment Variables* below for a description of this file). Messages remain in this file until specifically removed.

On the command line, *options* start with a dash (—) and any other arguments are taken to be destinations (recipients). If no recipients are specified, *mailx* will attempt to read messages from the *mailbox*. Command line options are:

- e Test for presence of mail. The command *mailx* prints nothing and exits with a successful return code if there is mail to read.
- f [*filename*]  
Read messages from *filename* instead of *mailbox*. If no *filename* is specified, the *mailbox* is used.
- F Record the message in a file named after the first recipient. Overrides the *record* variable, if set (see *Environment Variables*).

PI UN

—h*number*  
The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.

- H Print header summary only.
- i Ignore receipt of SIGINT. See also *ignore* (*Environment Variables*).
- n Do not initialise from the system default *mailx.rc* file.
- N Do not print initial header summary.

PI UN

—r *address*  
Pass *address* to network delivery software. All tilde commands are disabled.

- s *subject*  
Set the Subject header field to *subject*.
- u *user*  
Read *user's mailbox*. This is only effective if *user's mailbox* is not read protected.



When reading mail, *mailx* is in *command mode*. A header summary of the first several messages is displayed, followed by a prompt indicating *mailx* can accept regular commands (see **Commands** below). When sending mail, *mailx* is in *input mode*. If no subject is specified on the command line, a prompt for the subject is printed. As the message is typed, *mailx* will read the message and store it in a temporary file. Commands may be entered by beginning a line with the tilde (~) escape character followed by a single command letter and optional arguments. See **Tilde Escapes** for a summary of these commands.

At any time, the behaviour of *mailx* is governed by a set of *environment variables*. These are flags and valued parameters which are set and cleared via the *set* and *unset* commands. See **Environment Variables** below for a summary of these parameters.

Regular commands are of the form:

[ command ] [ msglist ] [ arguments ]

If no command is specified in *command mode*, *print* is assumed. In *input mode*, commands are recognised by the escape character, and lines not treated as commands are taken as input for the message.

Each message is assigned a sequential number, and there is at any time the notion of a "current" message, marked by a > in the header summary. Many commands take an optional list of messages (*msglist*) to operate on, which defaults to the current message. A *msglist* is a list of message specifications separated by spaces, which may include:

- n* Message number *n*.
- .* The current message.
- ^* The first undeleted message.
- \$* The last message.
- \** All messages.
- n-m* An inclusive range of message numbers.
- user* All messages from *user*.
- /string* All messages with *string* in the subject line (case ignored).
- :c* All messages of type *c*, where *c* is one of:
  - e* deleted messages
  - n* new messages
  - o* old messages
  - r* read messages
  - u* unread messages

Note that the context of the command determines whether this type of message specification makes sense.

Other arguments are usually arbitrary strings whose usage depends on the command involved. File names, where expected, can be specified with metacharacters understood by the command interpreter. Special characters are recognised by certain commands and are documented with the commands below.

At start-up time, *mailx* reads commands from a system-wide file to initialise certain parameters, then from a private start-up file (*\$HOME/.mailrc*) for personalised variables. Most regular commands are legal inside start-up files, the most common use being to set up initial display options and alias lists. The following commands are not legal in the start-up file: *!*, *Copy*, *edit*, *followup*, *Followup*, *hold*, *mail*, *preserve*, *reply*, *Reply*, *shell*, and *visual*. Any errors in the start-up file cause the remaining lines in the file to be ignored.

### Commands

The following is a complete list of *mailx* commands, most of which can be abbreviated. For each command, the abbreviated form is shown underneath the full form.

*!shell-command*

Escape to the command interpreter. See *SHELL* (Environment Variables).

*# comment*

Null command (comment). This may be useful in *.mailrc* files.

*=* Print the current message number.

*?* Prints a summary of commands.

*alias alias name...*

*a alias name ...*

*group alias name ...*

*g alias name ...*

Declare an alias for the given *names*. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

*alternates [name ...]*

*alt [name ...]*

Declares a list of alternate names for the user's login. When responding to a message, these names are removed from the list of recipients for the response. With no arguments, *alternates* prints the current list of alternate names. See also *allnet* (Environment Variables).

*cd [directory]*

*chdir [directory]*

*ch [directory]*

Change directory. If *directory* is not specified, *\$HOME* is used.



copy [filename]

c [filename]

copy [msglist] filename

c [msglist] filename

Copy messages to the file *filename* without marking the messages as saved. Otherwise equivalent to the save command.

Copy [msglist]

C [msglist]

Save the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the Save command.

delete [msglist]

d [msglist]

Delete messages from the *mailbox*. If *autoprint* is set, the next message after the last one deleted is printed (see Environment Variables).

discard [header-field ...]

di [header-field ...]

ignore [header-field ...]

ig [header-field ...]

Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are *status* and *cc*. The fields are included when the message is saved. The *Print* and *Type* commands override this command.

dp [msglist]

dt [msglist]

Delete the specified messages from the *mailbox* and print the next message after the last one deleted. Roughly equivalent to a *delete* command followed by a *print* command.

echo string ...

ec string ...

Echo the given strings (like *echo(1)*).

edit [msglist]

e [msglist]

Edit the given messages. The messages are placed in a temporary file and the *EDITOR* variable is used to get the name of the editor (see Environment Variables). Default editor is *ed*.

exit

ex

xit

x

Exit from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also *quit*).

**file** [*filename*]

**fi** [*filename*]

**folder** [*filename*]

**fold** [*filename*]

Quit from the current file of messages and read in the file *filename*. Several special characters are recognised when used as *filenames*, with the following substitutions:

%	the current <i>mailbox</i> .
%user	the <i>mailbox</i> for user.
#	the previous file.
&	the current <i>mbox</i> .

Default file is the current *mailbox*.

**folders**

Print the names of the files in the directory set by the *folder* variable (see *Environment Variables*).

**followup** [*message*]

**fo** [*message*]

Respond to a message, recording the response in a file whose name is derived from the author of the message. Overrides the *record* variable, if set. See also the Followup, Save, and Copy commands and *outfolder* (*Environment Variables*).

**Followup** [*msglist*]

**F** [*msglist*]

Respond to the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line is taken from the first message and the response is recorded in a file whose name is derived from the author of the first message. See also the followup, Save, and Copy commands and *outfolder* (*Environment Variables*).

**from** [*msglist*]

**f** [*msglist*]

Prints the header summary for the specified messages.

**group** *alias name* ...

**g** *alias name* ...

**alias** *alias name* ...

**a** *alias name* ...

Declare an alias for the given *names*. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.



**headers** [*message*]

**h** [*message*]

Prints the page of headers which includes the message specified. The *screen* variable sets the number of headers per page (see Environment Variables). See also the *z* command.

**help** Prints a summary of commands.

**hold** [*msglist*]

**ho** [*msglist*]

**preserve** [*msglist*]

**pre** [*msglist*]

Holds the specified messages in the *mailbox*.

**if** *s*|*r*

*mail-command* . . .

**else** *mail-command* . . .

**endif**

**i** *s*|*r*

*mail-command* . . .

**el** *mail-command* . . .

**en**

Conditional execution, where *s* will execute following *mail-commands*, up to an *else* or *endif*, if the program is in *send* mode, and *r* causes the *mail-commands* to be executed only in *receive* mode. Useful in the *.mailrc* file.

**ignore** [*header-field* ...]

**ig** [*header-field* ...]

**discard** [*header-field* ...]

**di** [*header-field* ...]

Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are *status* and *cc*. All fields are included when the message is saved. The *Print* and *Type* commands override this command.

**list**

**l** Prints all commands available. No explanation is given.

**mail** *name* ...

**m** *name* ...

Mail a message to the specified users.

**mbox** [*msglist*]

**mb** [*msglist*]

Arrange for the given messages to end up in the standard *mbox* save file when *mailx* terminates normally. See *MBOX* (Environment Variables) for a description of this file. See also the *exit* and *quit* commands.

**next** [*message*]

**n** [*message*]

Go to next message matching *message*. A *msglist* may be specified, but in this case the first valid message in the list is the only one used. This is useful for jumping to the next message from a specific user, since the name would be taken as a command in the absence of a real command. See the discussion of *msglists* above for a description of possible message specifications.

**pipe** [*msglist*] [*shell-command*]

**pi** [*msglist*] [*shell-command*]

**|** [*msglist*] [*shell-command*]

Pipe the message through the given *shell-command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the *cmd* variable. If the *page* variable is set, a form feed character is inserted after each message (see Environment Variables).

**preserve** [*msglist*]

**pre** [*msglist*]

**hold** [*msglist*]

**ho** [*msglist*]

Preserve the specified messages in the *mailbox*.

**Print** [*msglist*]

**P** [*msglist*]

**Type** [*msglist*]

**T** [*msglist*]

Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the *ignore* command.

**print** [*msglist*]

**p** [*msglist*]

**type** [*msglist*]

**t** [*msglist*]

Print the specified messages. If *crt* is set, the messages longer than the number of lines specified by the *crt* variable are paged through the command specified by the *PAGER* environment variable. The default command is *pg*. (See Environment Variables.)

**quit**

**q**

Exit from *mailx*, storing messages that were read in *mbox* and unread messages in the *mailbox*. Messages that have been explicitly saved in a file are deleted.



Reply [*msglist*]

Respond [*msglist*]

R [*msglist*]

Send a response to the author of each message in the *msglist*. The subject line is taken from the first message. If *record* is set to a file name, the response is saved at the end of that file (see *Environment Variables*).

reply [*message*]

r [*message*]

Reply to the specified *message*, including all other recipients of the message. If *record* is set to a file name, the response is saved at the end of that file (see *Environment Variables*).

Save [*msglist*]

S [*msglist*]

Save the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the *Copy*, *followup*, and *Followup* commands and *outfolder* (*Environment Variables*).

save [*filename*]

s [*filename*]

save [*msglist*] *filename*

s [*msglist*] *filename*

Save the specified messages in the given file. The file is created if it does not exist. The message is deleted from the *mailbox* when *mailx* terminates unless *keepsave* is set (see also *Environment Variables* and the *exit* and *quit* commands).

set [*name*]

se [*name*]

set [*name=string*]

se [*name=string*]

set [*name=number*]

se [*name=number*]

Define a variable called *name*. The variable may be given a null, string, or numeric value. Set by itself prints all defined variables and their values. See *Environment Variables* for detailed descriptions of the *mailx* variables.

shell

sh Invoke an interactive command interpreter (see also *SHELL* (*Environment Variables*)).

size [*msglist*]

si [*msglist*]

Print the size in characters of the specified messages.

**source** *filename*

**so** *filename*

Read commands from the given file and return to command mode.

**top** [*msglist*]

**to** [*msglist*]

Print the top few lines of the specified messages. If the *toplines* variable is set, it is taken as the number of lines to print (see **Environment Variables**). The default is 5.

**touch** [*msglist*]

**tou** [*msglist*]

Touch the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the *mbox* upon normal termination. See **exit** and **quit**.

**Type** [*msglist*]

**T** [*msglist*]

**Print** [*msglist*]

**P** [*msglist*]

Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the *ignore* command.

**type** [*msglist*]

**t** [*msglist*]

**print** [*msglist*]

**p** [*msglist*]

Print the specified messages. If *crt* is set, the messages longer than the number of lines specified by the *crt* variable are paged through the command specified by the *PAGER* variable. The default command is *pg*. (See **Environment Variables**).

**undelete** [*msglist*]

**u** [*msglist*]

Restore the specified deleted messages. Will only restore messages deleted in the current mail session. If *autoprint* is set, the last message of those restored is printed (see **Environment Variables**).

**unset** *name* ...

**uns** *name* ...

Causes the specified variables to be erased. If the variable was imported from the execution environment (i.e., an environment variable) then it cannot be erased.

mv

**version**

**ve** ... Prints the current version and release date.



**visual** [*msglist*]

**v** [*msglist*]

Edit the given messages with a screen editor. The messages are placed in a temporary file and the *VISUAL* variable is used to get the name of the editor (see *Environment Variables*).

**write** [*msglist*] *filename*

**w** [*msglist*] *filename*

Write the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the *save* command.

**xit**

**exit**

**x** Exit from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also *quit*).

**z**[+|-]

Scroll the header display forward or backward one screen-full. The number of headers displayed is set by the *screen* variable (see *Environment Variables*).

### Tilde Escapes

The following commands may be entered only from *input mode*, by beginning a line with the tilde escape character (~). See *escape* (*Environment Variables*) for changing this special character.

~! *shell-command*

Escape to the command interpreter.

~ Simulate end of file (terminate message input).

~: *mail-command*

~\_ *mail-command*

Perform the command-level request. Valid only when sending a message while reading mail.

~? Print a summary of tilde escapes.

~A Insert the autograph string *Sign* into the message (see *Environment Variables*).

~a Insert the autograph string *sign* into the message (see *Environment Variables*).

~b *name* ...

Add the *name* to the blind carbon copy (Bcc) list.

~c *name* ...

Add the *name* to the carbon copy (Cc) list.

~d Read in the *dead.letter* file. See *DEAD* (*Environment Variables*) for a description of this file.

~e Invoke the editor on the partial message. See also *EDITOR* (*Environment Variables*).

**~f [msglist]**

Forward the specified messages. The messages are inserted into the message, without alteration.

**~h**

Prompt for Subject line and To, Cc, and Bcc lists. If the field is displayed with an initial value, it may be edited as if it had just been typed.

**~i string**

Insert the value of the named variable into the text of the message. For example, `~A` is equivalent to `~i Sign`.

**~m [msglist]**

Insert the specified messages into the letter, shifting the new text to the right one tab stop. Valid only when sending a message while reading mail.

**~p**

Print the message being entered.

**~q**

Quit from input mode by simulating an interrupt. If the body of the message is not null, the partial message is saved in *dead.letter*. See *DEAD (Environment Variables)* for a description of this file.

**~r filename****~< filename****~<!shell-command**

Read in the specified file. If the argument begins with an exclamation point (!), the rest of the string is taken as an arbitrary system command and is executed, with the standard output inserted into the message.

**~s string ...**

Set the subject line to *string*.

**~t name ...**

Add the given *name* to the To list.

**~v**

Invoke a preferred screen editor on the partial message. See also *VISUAL (Environment Variables)*.

**~w filename**

Write the partial message onto the given file, without the header.

**~x**

Exit as with `~q` except the message is not saved in *dead.letter*.

**~ shell-command**

Pipe the body of the message through the given *shell-command*. If the *shell-command* returns a successful exit status, the output of the *shell-command* replaces the message.

**Environment Variables**

The following are environment variables taken from the execution environment and are not alterable within *mailx*.

**HOME=directory**

The user's base of operations.



**MAILRC=filename**

The name of the start-up file. Default is *\$HOME/.mailrc*.

The following variables are internal *mailx* variables. They may be imported from the execution environment or set via the *set* command at any time. The *unset* command may be used to erase variables.

**allnet**

All network names whose last component (login name) match are treated as identical. This causes the *msglist* message specifications to behave similarly. Default is *noallnet*. See also the *alternates* command and the *metoo* variable.

**append**

Upon termination, append messages to the end of the *mbox* file instead of prepending them. Default is *noappend*.

**askcc**

Prompt for the Cc list after message is entered. Default is *noaskcc*.

**asksub**

Prompt for subject if it is not specified on the command line with the *—s* option. Enabled by default.

**autoprint**

Enable automatic printing of messages after *delete* and *undelete* commands. Default is *noautoprint*.

**bang**

Enable the special-case treatment of exclamation points (!) in escape command lines as in *vi*(1). Default is *nobang*.

**cmd=shell-command**

Set the default command for the *pipe* command. No default value.

**conv=conversion**

Convert *uucp* addresses to the specified address style. Conversion is disabled by default. See also *sendmail* and the *—U* command line option.

**crt=number**

Pipe messages having more than *number* lines through the command specified by the value of the *PAGER* variable (*pg*(1) by default). Disabled by default.

**DEAD=filename**

The name of the file in which to save partial letters in case of untimely interrupt or delivery errors. Default is *\$HOME/dead.letter*.

**debug**

Enable verbose diagnostics for debugging. Messages are not delivered. Default is *nodebug*.

**dot**

Take a period on a line by itself during input from a terminal as end-of-file. Default is *nodot*.

**EDITOR**=*shell-command*

The command to run when the *edit* or *~e* command is used. Default is *ed(1)*.

**escape**=*c*

Substitute *c* for the *~* escape character.

**folder**=*directory*

The directory for saving standard mail files. User-specified file names beginning with a plus (+) are expanded by preceding the file name with this directory name to obtain the real file name. If *directory* does not start with a slash (/), *\$HOME* is prepended to it. In order to use the plus (+) construct on a *mailx* command line, *folder* must be an exported environment variable. There is no default for the *folder* variable. See also *outfolder* below.

**header**

Enable printing of the header summary when entering *mailx*. Enabled by default.

**hold** Preserve all messages that are read in the *mailbox* instead of putting them in the standard *mbox* save file. Default is *nohold*.

**ignore**

Ignore receipt of **SIGINT** while entering messages. Handy for noisy dial-up lines. Default is *noignore*.

**ignoreeof**

Ignore end-of-file during message input. Input must be terminated by a period (.) on a line by itself or by the *~.* command. Default is *noignoreeof*. See also *dot* above.

**keep**

When the *mailbox* is empty, truncate it to zero length instead of removing it. Disabled by default.

**keepsave**

Keep messages that have been saved in other files in the *mailbox* instead of deleting them. Default is *nokeepsave*.

**MBOX**=*filename*

The name of the file to save messages which have been read. The *xit* command overrides this function, as does saving the message explicitly in another file. Default is *\$HOME/mbox*.

**metoo**

If the user's login appears as a recipient, do not delete it from the list. Default is *nometoo*.

**LISTER**=*shell-command*

The command (and options) to use when listing the contents of the *folder* directory. The default is *ls*.



**onehop**

When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away).

**outfolder**

Causes the files used to record outgoing messages to be located in the directory specified by the *folder* variable unless the path name is absolute. Default is *nooutfolder*. See *folder* above and the *Save*, *Copy*, *followup*, and *Followup* commands.

**page** Used with the *pipe* command to insert a form feed after each message sent through the pipe. Default is *nopage*.

**PAGER=shell-command**

The command to use as a filter for paginating output. This can also be used to specify the options to be used. Default is *pg*.

**prompt=string**

Set the *command mode* prompt to *string*. Default is *?*.

**quiet**

Refrain from printing the opening message and version when entering *mailx*. Default is *noquiet*.

**record=filename**

Record all outgoing mail in *filename*. Disabled by default. See also *outfolder* above.

**save** Enable saving of messages in *dead.letter* on interrupt or delivery error. See *DEAD* for a description of this file. Enabled by default.

**screen=number**

Sets the number of lines in a screen-full of headers for the *headers* command.

**sendmail=shell-command**

Alternate command for delivering messages. Default is *mail*.

**sendwait**

Wait for background mailer to finish before returning. Default is *nosendwait*.

**SHELL=shell-command**

The name of a preferred command interpreter. Default is *sh*.

**showto**

When displaying the header summary, and the message is from the user, print the recipient's name instead of the author's name.

**sign=string**

The variable inserted into the text of a message when the `~a` (autograph) command is given. No default (see also `~i` (Tilde Escapes)).

**Sign=string**

The variable inserted into the text of a message when the `~A` command is given. No default (see also `~i` (Tilde Escapes)).

**toplines=number**

The number of lines of header to print with the `to` command. Default is 5.

**VISUAL=shell-command**

The name of a preferred screen editor. Default is system-dependent; usually `vi`.

## FILES

<code>\$HOME/.mailrc</code>	user's start-up file
<code>mailx.rc</code>	system-wide start-up file.
<code>\$HOME/mbox</code>	secondary storage file
<code>dead.letter</code>	unmailable text

## SEE ALSO

`mail(1)`, `pg(1)`, `ls(1)`, `vi(1)`.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

This utility is not optional in the SVID.

In the description of the `—i` option and the *ignore* environment variables, the words "Ignore interrupts" have been changed to "Ignore receipt of SIGINT".

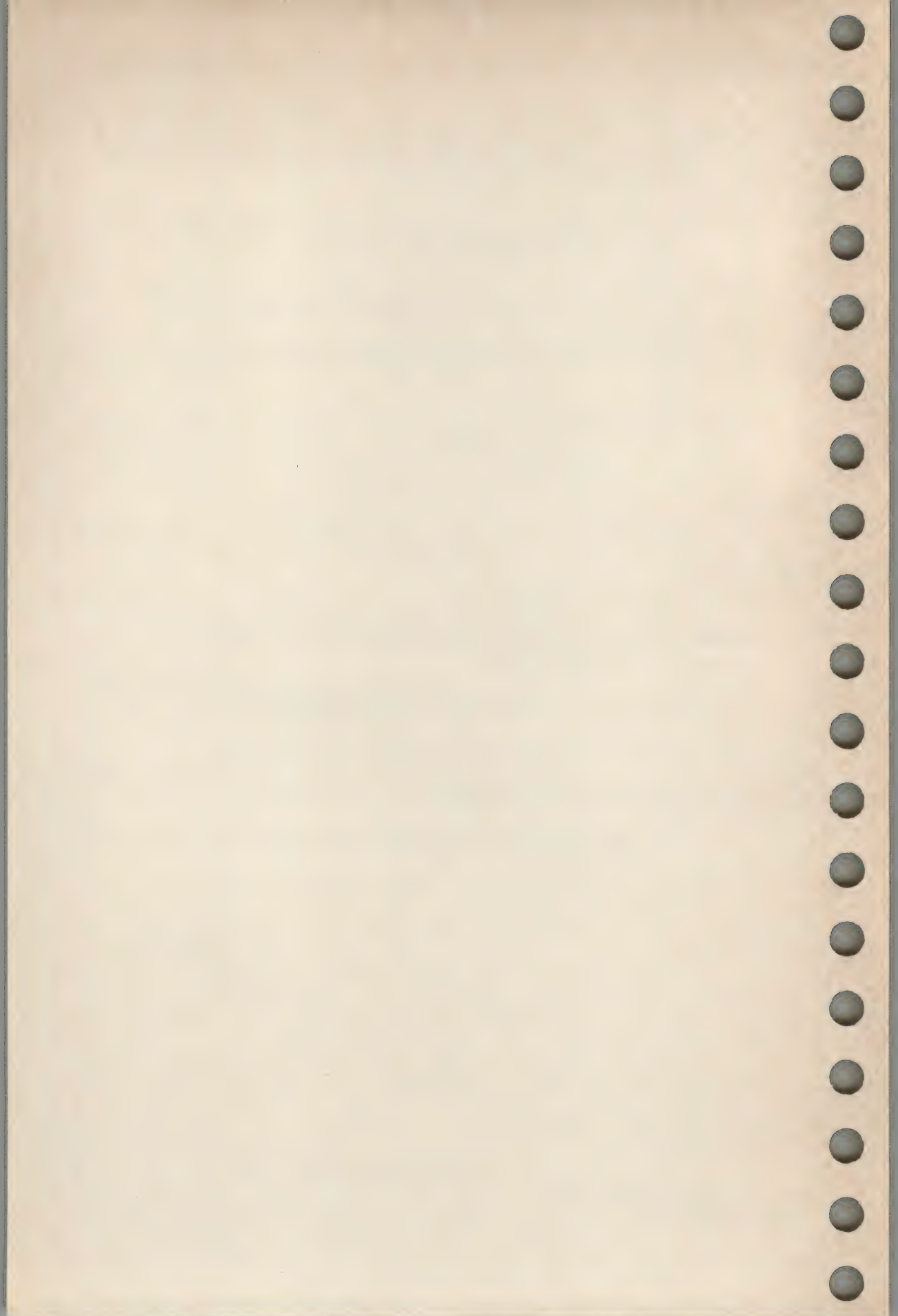
The word "*filename*" has been added to the description of the *copy* option.

In the description of the *fold* command, the words "read in the specified file" have been changed to "read in the file *filename*".

In the Tilde Escapes section, in the description of the `~i` option, the words "*~i Sign*" have been added to complete the sentence.

A space has been inserted for the `—r`, `—s` and `—u` options.





## NAME

make — maintain, update, and regenerate groups of programs

## SYNOPSIS

make [—f *makefile*] [—p] [—i] [—k] [—s] [—r] [—n] [—e] [—t] [—q]  
[ *name* ... ]

## DESCRIPTION

The options are interpreted as follows:

—f *makefile*

Description file name. The argument *makefile* is assumed to be the name of a description file. A file name of — denotes the standard input.

UN

## —p Print out the complete set of macro definitions and target descriptions.

—i Ignore error codes returned by invoked commands. This mode is entered if the fake target name *.IGNORE* appears in the description file.

—k Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

—s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name *.SILENT* appears in the description file.

—r Do not use the built-in rules.

—n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ are printed.

—e Environment variables override assignments within makefiles.

—t Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

—q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

The following target names may be defined in the *makefile*, and are interpreted as follows:

*.DEFAULT*

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name *.DEFAULT* are used if it exists.

*.PRECIOUS*

Dependents of this target will not be removed if SIGINT or SIGQUIT occur.

*.SILENT*

Same effect as the —s option.

*.IGNORE*

Same effect as the —i option.



The command *make* executes commands in *makefile* to update one or more target *names*. The argument *name* is typically a program, but may also be a macro definition, see **Environment**.

If no *-f* option is present, *makefile*, *Makefile*, and the SCCS files *s.makefile* and *s.Makefile* are tried in order. If *makefile* is *—*, the standard input is used. More than one *-f makefile* argument pair may appear.

The command *make* updates a target only if its dependents are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out-of-date.

The argument *makefile* contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a *:*, then a (possibly null) list of prerequisite files or dependencies. Text following a *;* and all following lines that begin with a tab are commands to be executed to update the target. The first line that does not begin with a tab or *#* begins a new dependency or macro definition. To continue commands across more than one line, all but the last line must be terminated by a backslash. Everything printed by *make* (except the initial tab) is passed directly to the command interpreter unchanged.

The symbols *#* and *<newline>* surround comments.

The following *makefile* says that *pgm* depends on two files *a.o* and *b.o*, and that they in turn depend on their corresponding source files (*a.c* and *b.c*) and a common file *incl.h*:

```
pgm: a.o b.o
    cc a.o b.o -o pgm
a.o: incl.h a.c
    cc -c a.c
b.o: incl.h b.c
    cc -c b.c
```

Command lines are executed one at a time. The first one or two characters in a command can be the following: *-*, *@*, *-@*, or *@-*. If *@* is present, printing of the command is suppressed. If *-* is present, *make* ignores an error. A line is printed when it is executed unless the *-s* option is present, or the entry *.SILENT:* is in *makefile*, or unless the initial character sequence contains a *@*. The *-n* option specifies printing without execution; however, if the command line has the string *\$(MAKE)* in it, the line is always executed (see discussion of the *MAKEFLAGS* macro under **Environment**). The *-t* (touch) option updates the modified date of a file without executing any commands.

Each command line is executed in a separate invocation of the command interpreter.

Commands returning non-zero status normally terminate *make*. If the *—i* option is present, or the entry *.IGNORE:* appears in *makefile*, or the initial character sequence of the command contains *—*, the error is ignored. If the *—k* option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the special name *.PRECIOUS*.

#### Environment

The environment is read by *make*. All variables are assumed to be macro definitions and are processed as such. Macro definitions may also occur on the command line and in the *makefile*. Certain macros have internal defaults within *make*. The order of precedence depends on the setting of the *—e* option, as follows:

Without *—e*:

1. command line
2. *makefile*
3. environment
4. internal

With *—e*:

1. command line
2. environment
3. *makefile*
4. internal

The environment variable *MAKEFLAGS* is processed by *make* as containing any legal input option (except *—f* and *—p*) defined for the command line. Further, upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, *MAKEFLAGS* always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the *—n* option is used, the command *\$(MAKE)* is executed anyway; hence, one can perform a *make —n* recursively on a whole software system to see what would have been executed. This is because the *—n* is put in *MAKEFLAGS* and passed to further invocations of *\$(MAKE)*. This is one way of debugging all of the *makefiles* for a software project without actually doing anything.

#### Macros

Entries of the form *string1 = string2* are macro definitions. The macro *string2* is defined as all characters up to a comment character or an unescaped newline. Subsequent appearances of *\$(string1[:subst1=[subst2]])* are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional *:subst1=subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, newline characters, and beginnings of



lines. An example of the use of the substitute sequence is shown under Libraries.

### Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

- \$\*** The **\$\*** macro stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.
- \$@** The **\$@** macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.
- \$<** The **\$<** macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out-of-date with respect to the target (i.e., the "manufactured" dependent file name). Thus, in the **.c.o** rule, the **\$<** macro would evaluate to the **.c** file. An example for making optimised **.o** files from **.c** files is:

```
.c.o:
    cc -c -O $*.c
```

or:

```
.c.o:
    cc -c -O $<
```

- \$?** The **\$?** macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out-of-date with respect to the target; essentially, those modules which must be rebuilt.
- \$%** The **\$%** macro is only evaluated when the target is an archive library member of the form **lib.a(file.o)**. In this case, **\$@** evaluates to **lib.a** and **\$%** evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros, the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **\$(@D)** refers to the directory part of the string **\$@**. If there is no directory part, **./** is generated. The only macro excluded from this alternative form is **\$?**.

### Suffixes

Certain names (for instance, those ending with **.o**) have inferable prerequisites such as **.c**, **.s**, etc.. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. Inference rules in the makefile override the default rules.

The internal rules for *make* are compiled into the *make* program. To print out the rules compiled into the *make* program, the following command is used:

```
make —fp — 2>/dev/null </dev/null
```

A tilde in the above rules refers to an SCCS file. Thus, the rule *.c.o* would transform an SCCS C source file into an object file (*.o*). Because the *s.* of the SCCS files is a prefix, it is incompatible with *make*'s suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (e.g., *.c:*) is the definition of how to build *x* from *x.c*. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., command procedures, simple C programs).

Additional suffixes are given as the dependency list for *.SUFFIXES*. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; *SUFFIXES:* with no dependencies clears the list of suffixes.

#### Inference Rules

The first example can be done more briefly.

```
pgm: a.o b.o
      cc a.o b.o —o pgm
a.o b.o: incl.h
```

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, *CFLAGS*, *LFLAGS* and *YFLAGS* are used for compiler options to *cc*, *lex* and *yacc*, respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix *.o* from a file with suffix *.c* is specified as an entry with *.c.o:* as the target and no dependents. Commands associated with the target define the rule for making a *.o* file from a *.c* file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.



## Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus *lib.a(file.o)* and *\$(LIB)(file.o)* both refer to an archive library which contains *file.o*. (This assumes the *LIB* macro has been previously defined.) The expression *\$(LIB)(file1.o file2.o)* is not legal. Rules pertaining to archive libraries have the form *.XX.a* where the *XX* is the suffix from which the archive member is to be made. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib.a:    lib.a(file1.o) lib.a(file2.o) lib.a(file3.o)
          @echo lib.a is now up-to-date

.c.a:
          $(CC) -c $(CFLAGS) $<
          ar rv $@ $*.o
          rm -f $*.o
```

In fact, the *.c.a* rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited, example of an archive library maintenance construction follows:

```
lib.a:    lib.a(file1.o) lib.a(file2.o) lib.a(file3.o)
          $(CC) -c $(CFLAGS) $(?:.o=.c)
          ar rv lib.a $?
          rm $?
          @echo lib.a is now up-to-date

.c.a:;
```

Here the substitution mode of the macro expansions is used. The *\$?* list is defined to be the set of object file names (inside *lib.a*) whose C source files are out-of-date. The substitution mode translates the *.o* to *.c*. Note also the disabling of the *.c.a:* rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

## FILES

```
[Mm]akefile    rules file
s.[Mm]akefile  SCCSed rules file
```

## SEE ALSO

cc(1D), lex(1D), sh(1), yacc(1D).

## APPLICATION USAGE

The characters = : and @ in file names may give trouble.

It is guaranteed that the appropriate transformations will be built into *make* on all X/OPEN systems for software developers using the C programming language, *lex*, *yacc*, shell programs and archives. The rules will also cater for the use of *SCCS*.

CHANGE HISTORY

First released in Issue 2.

Issue 2

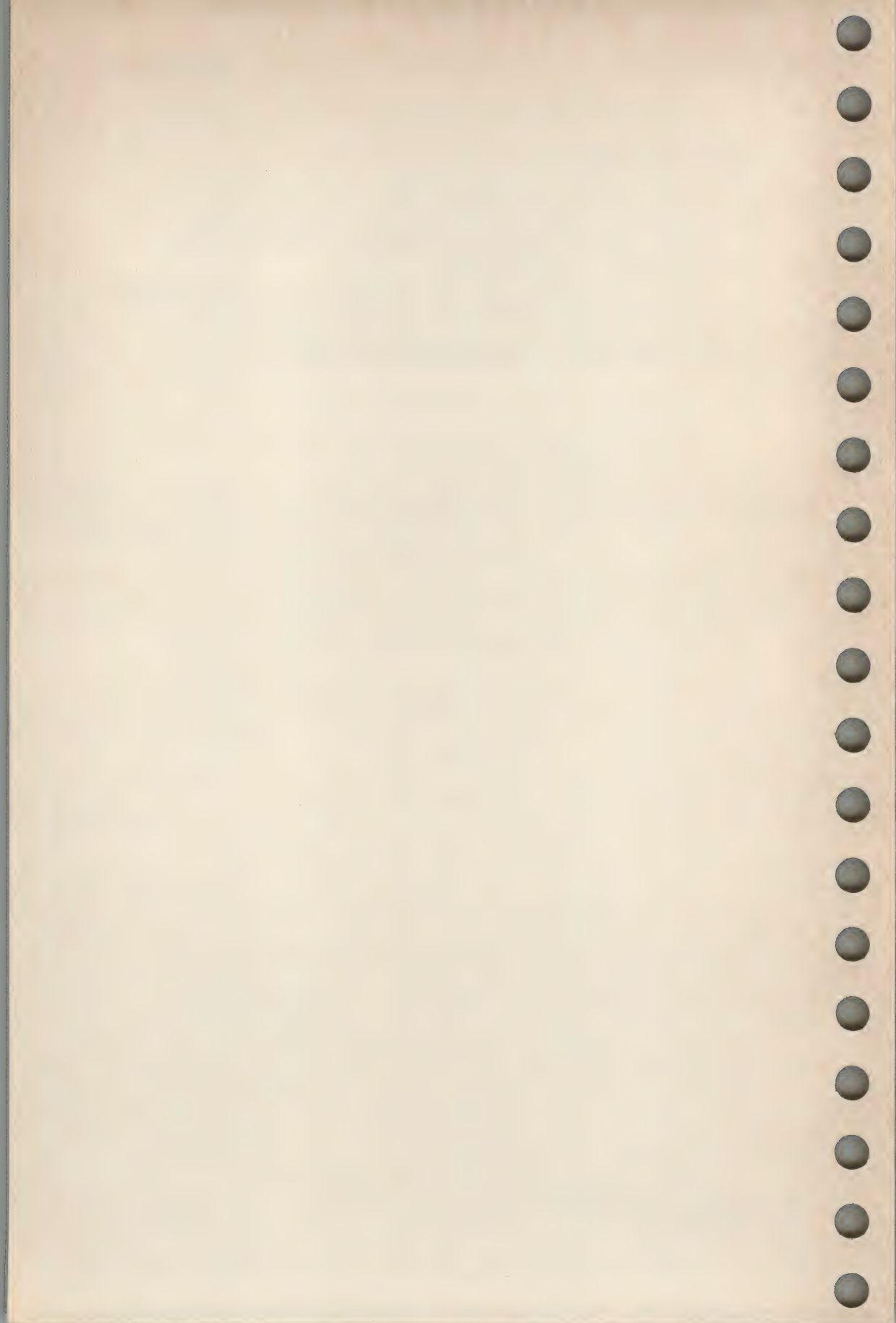
Derived from the entry in Issue 2 of the SVID with the following changes:

The paragraph "Each command line is executed in a separate invocation of the command interpreter." has been added.

An erroneous *B* has been removed from the names of the *CFLAGS*, *LFLAGS* and *YFLAGS* macros.

The order of precedence of macro settings has been added.





## NAME

`mesg` — permit or deny messages

## SYNOPSIS

`mesg [y|n]`

## DESCRIPTION

The command `mesg` with argument `n` prevents another user from writing to the invoking user's terminal (e.g., by using `write`, see `write(1)`). The command `mesg` with argument `y` reinstates write permission.

OF

With no arguments, `mesg` reports the current state without changing it.

## EXIT STATUS

Exit status is

- 0 messages are receivable
- 1 messages are not receivable
- 2 error

## SEE ALSO

`wall(1)`, `write(1)`.

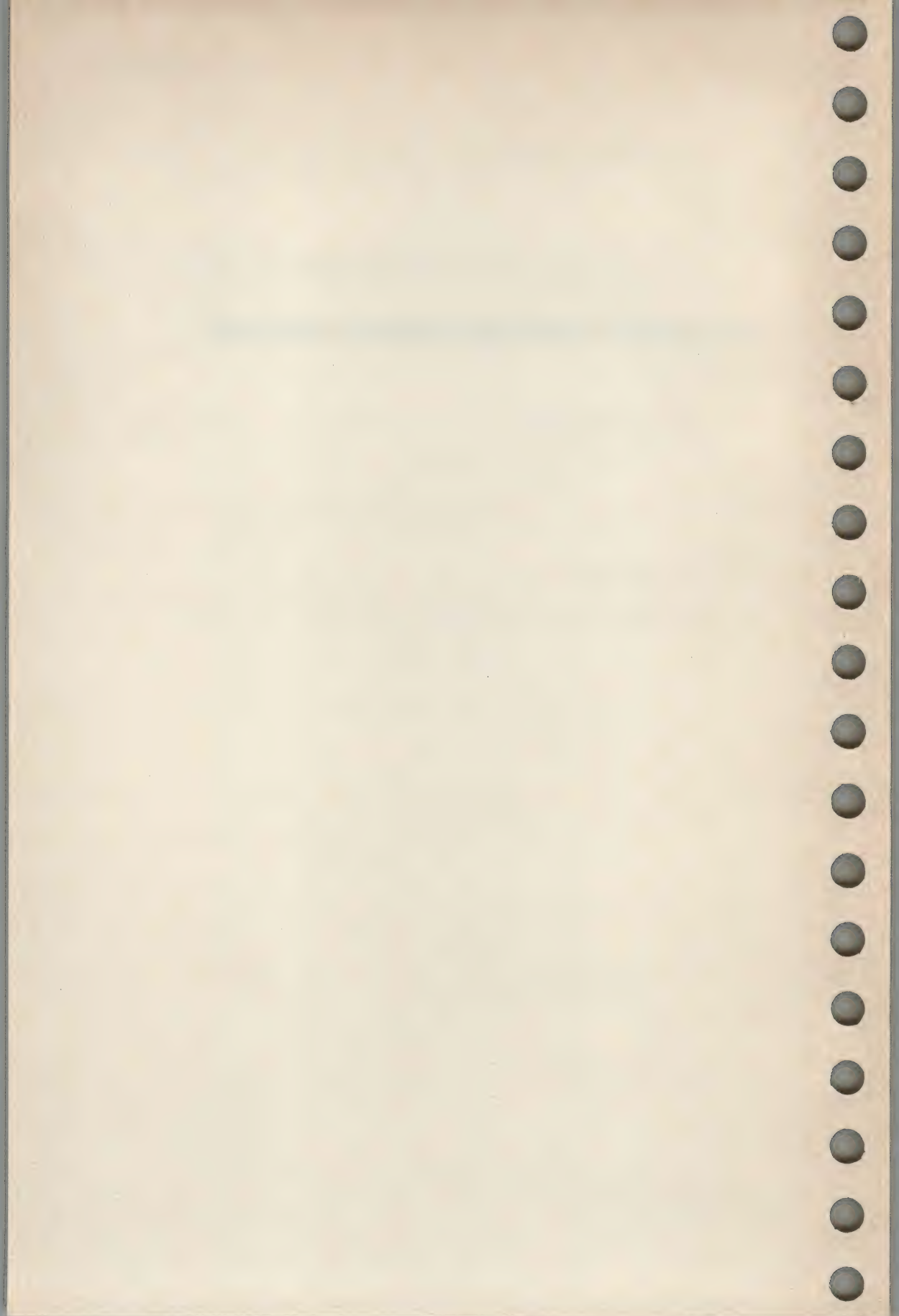
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`mkdir` — make a directory

## SYNOPSIS

`mkdir dirname...`

## DESCRIPTION

The command `mkdir` creates the specified directories. Standard entries, `.`, for the directory itself, and `..`, for its parent, are made automatically.

The command `mkdir` requires write permission in the parent directory.

## EXIT STATUS

Exit status is

0	all directories were successfully made
non-0	otherwise

## SEE ALSO

`rm(1)`.

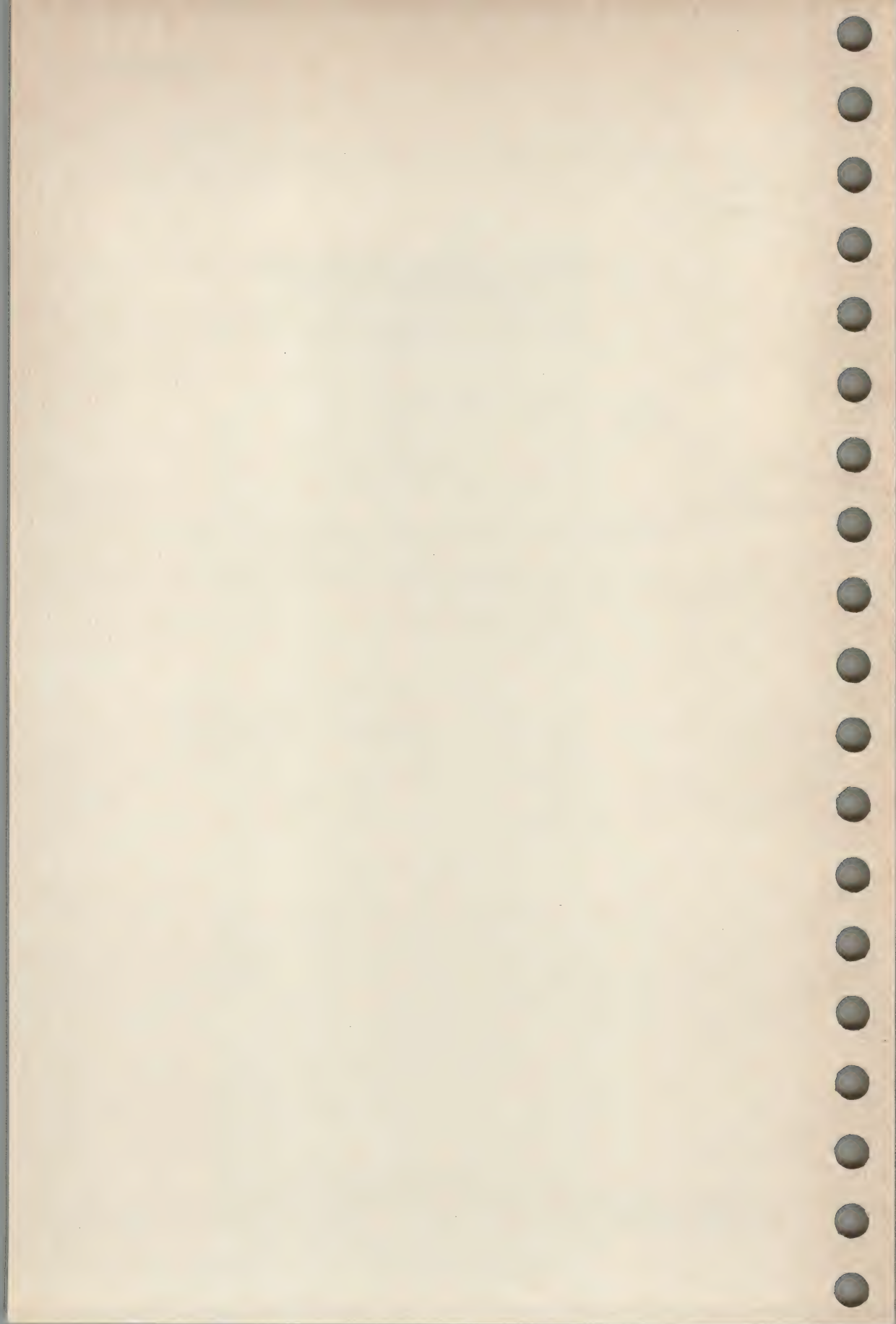
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

mknod — build fifo file (OPTIONAL)

## SYNOPSIS

mknod name p

## DESCRIPTION

The command *mknod* creates a fifo (named pipe).

The first argument is the *name* of the entry. The second argument must be the letter *p* to denote that a fifo is to be created.

## SEE ALSO

mknod(2).

## APPLICATION USAGE

For licensing reasons, this command is not formally part of the X/OPEN interface definition. However, Application Developers may need to create fifos from an equivalent utility. The *mknod*(2) service is always present as part of the X/OPEN definition, but cannot be accessed directly from scripts of commands being interpreted by, for example, *sh*(1).

The following *mkpipe* program could be used in such circumstances.

```

/*
 * mkpipe:
 * Non-proprietary version of mknod(1) for the construction of fifos.
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

/*
 * cater for systems (e.g., System V) which do not have symbolic names
 * for permissions yet.
 */
#ifndef S_IRUSR
#   define S_IRUSR 0000400
#   define S_IWUSR 0000200
#   define S_IRGRP 0000040
#   define S_IWGRP 0000020
#   define S_IROTH 0000004
#   define S_IWOTH 0000002
#endif

/* mode for the FIFO */
#define RW_FIFO (S_IFIFO|S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

```



```
/*
 * Program is invoked with exactly one argument.
 * The argument is the pathname of the proposed fifo.
 * Creates a fifo with read+write permission for everyone.
 */

main(argc, argv, env)
int argc;
char **argv, **env;{

    if(argc != 2){
        fprintf(stderr,
            "mkpipe: ERROR: exactly one argument required\n");
        exit(1);
    }

    if(mknod(*+ +argv, RWFIFO, 0) == 0)
        exit(0);

    /*
     * Could not make the fifo.
     */
    fprintf(stderr,
        "mkpipe: ERROR: cannot make fifo %s\n", *argv);
    exit(2);
}
```

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID by taking a subset of the functionality described there.

The pathname */etc* has been removed from the SYNOPSIS section.

The example program has been added.

## NAME

`newgrp` — change to a new group (OPTIONAL)

## SYNOPSIS

`newgrp [—] [group]`

## DESCRIPTION

The command *newgrp* changes a user's group identification. The user remains logged in and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new real and effective group IDs.

Only exported environment variables retain their values after invoking *newgrp*. Otherwise, variables with a default value will be reset to that default, see *sh*(1).

With no arguments, *newgrp* changes the group identification back to the group specified in the user's password file entry.

The following statements are true only if the command interpreter named in the specified user's password file entry is *sh*, see *sh*(1). If the first argument to *newgrp* is a `—`, the environment will be changed to what would be expected if the user actually logged in again.

A password is demanded if the group has a password and the user does not, or if the group has a password and the user is not listed in */etc/group* as being a member of that group.

*Newgrp* will fail if the user is not listed in */etc/group* as being a member of that group and the group does not have a password.

## FILES

<i>/etc/group</i>	system's group file
<i>/etc/passwd</i>	system's password file

## SEE ALSO

*sh*(1), *exec*(2).

## APPLICATION USAGE

There is no convenient way to enter a password into */etc/group*. Use of group passwords is not encouraged, because by their very nature they encourage poor security practices. Group passwords may disappear in the future.

A common implementation of *newgrp* is that the current shell uses *exec* to overlay itself with *newgrp* which in turn overlays itself with a new shell after changing group. On some systems, however, this may not occur and *newgrp* may be invoked as a subprocess.

*Newgrp* is only intended for use from an interactive terminal. It does not offer a useful interface for the support of applications.



## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

This utility is not optional in the SVID.

The reference to passwords has been added.

A description of when *newgrp* could fail has been added.

## NAME

`news` — print news items (OPTIONAL)

## SYNOPSIS

`news [-a] [-n] [-s] [item ...]`

MV OF UN

## DESCRIPTION

The command `news` prints files from the system news directory.

When invoked without arguments, `news` prints the contents of all current files in the news directory, most recent first, with each preceded by an appropriate header. `News` stores the "currency" time as the modification date of a file named `.news_time` in the user's home directory (the identity of this directory is determined by the environment variable `HOME`); only files more recent than this currency time are considered current.

The `—a` option causes `news` to print all items, regardless of currency. In this case, the stored time is not changed.

The `—n` option causes `news` to report the names of the current items without printing their contents and without changing the stored time.

The `—s` option causes `news` to report how many current items exist, without printing their names or contents and without changing the stored time.

All other arguments are assumed to be specific news items that are to be printed.

If an interrupt signal is received during the printing of a news item, printing stops and the next item is started. Another interrupt within one second of the first causes the program to terminate.

## FILES

<code>/etc/profile</code>	system wide startup file
<code>\$HOME/.news_time</code>	stored "currency" time.

## CHANGE HISTORY

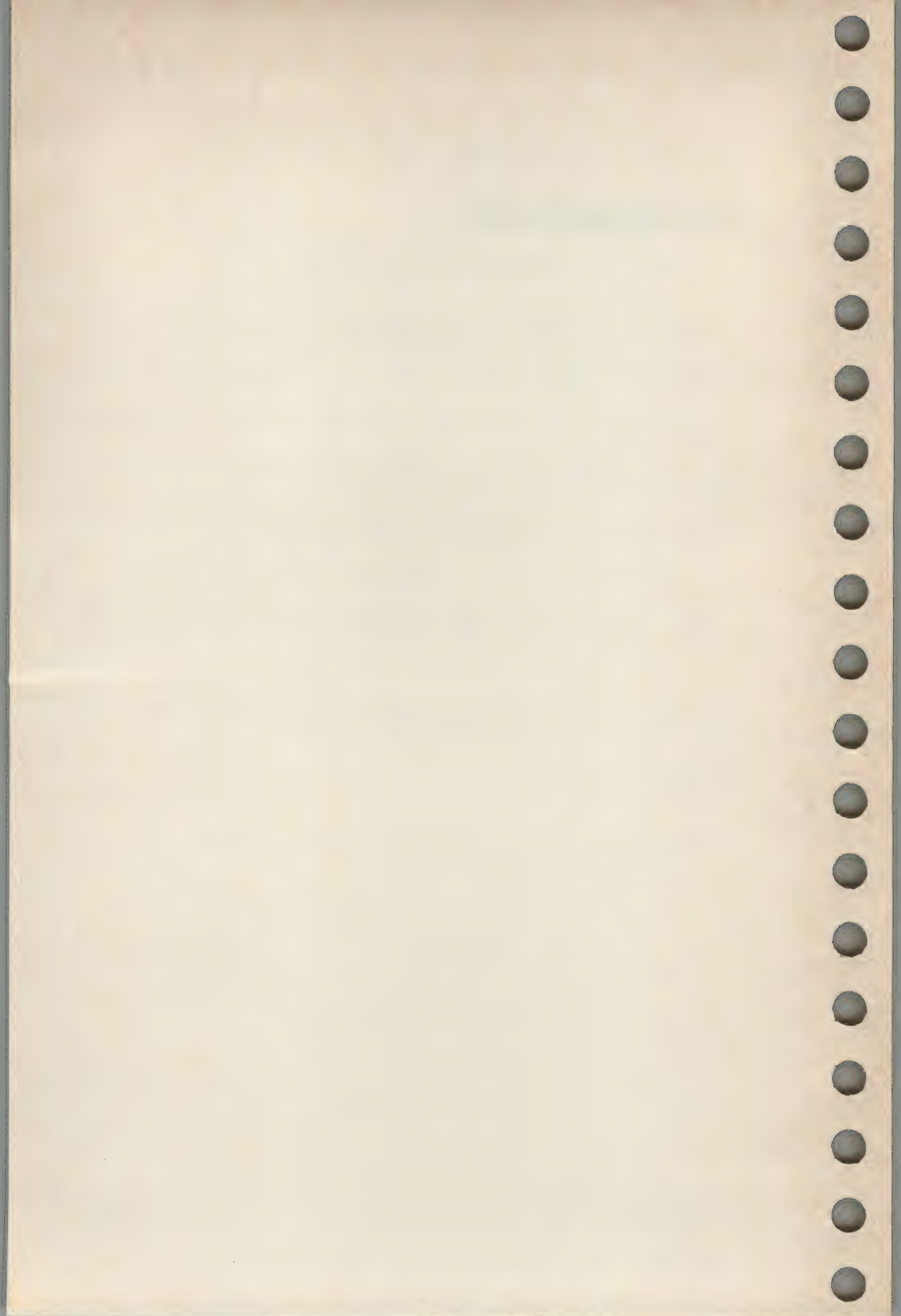
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

This utility is not optional in the SVID.





## NAME

*nl* — line numbering filter

## SYNOPSIS

*nl* [*file*]

*nl* [*—h**type*] [*—b**type*] [*—f**type*] [*—v**startnum*] [*—i**incr*] [*—p*] [*—l**num*]  
 [*—s**sep*] [*—w**width*] [*—n**format*] [*—d**delim*] [*file*]

UN

## DESCRIPTION

The command *nl* reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines on the standard output. Lines are numbered on the left. Additional functionality may be provided in accordance with the command options in effect.

*Nl* views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (e.g., no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signalled by input lines containing nothing but the following delimiter character(s):

<i>Line</i>	<i>Start of</i>
\\: \\: \\:	header
\\: \\:	body
\\:	footer

Unless otherwise specified, *nl* assumes the text being read is in a single logical page body.

Options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

**—b***type*

Specifies which logical page body lines are to be numbered. Recognised *types* and their meaning are:

**a**     number all lines;  
**t**     number lines with printable text only;  
**n**     no line numbering;

**p***string*

number only lines that contain the regular expression specified in *string*

Default *type* for logical page body is *t* (text lines numbered).

**—h***type*

Same as **—b***type* except for header. Default *type* for logical page header is *n* (no lines numbered).



- ftype**  
Same as **—btype** except for footer. Default for logical page footer is *n* (no lines numbered).
- p** Do not restart numbering at logical page delimiters.
- vstartnum**  
The initial value used to number logical page lines. Default is 1.
- incr**  
The increment value used to number logical page lines. Default is 1.
- ssep**  
The character(s) used in separating the line number and the corresponding text line. Default *sep* is a tab.
- width**  
The number of characters to be used for the line number. Default *width* is 6.
- nformat**  
The line numbering format. Recognised values are: *ln*, left justified, leading zeroes suppressed; *rn*, right justified, leading zeroes suppressed; *rz*, right justified, leading zeroes kept. Default *format* is *rn* (right justified).
- lnum**  
The number of blank lines to be considered as one. For example, **—l2** results in only the second adjacent blank being numbered (if the appropriate **—ha**, **—ba**, and/or **—fa** option is set). Default is 1.
- dxx** The delimiter characters specifying the start of a logical page section may be changed from the default characters (**\:**) to two user-specified characters. If only one character is entered, the second character remains the default character (**:**). No space should appear between the **—d** and the delimiter characters. To enter a backslash, two backslashes should be used.

#### EXAMPLE

The command:

```
nl —v10 —i10 —d!+ file1
```

will number *file1* starting at line number 10 with an increment of 10. The logical page delimiter is **"/+**.

#### SEE ALSO

**pr(1)**.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The words "Additional functionality may be provided" have been added to the first paragraph of the DESCRIPTION.

## NAME

`nm` — print name list of object file

## SYNOPSIS

`nm [ options ] file...`

## DESCRIPTION

The `nm` command displays the symbol table of each object file *file*. The argument *file* may be a relocatable or absolute object file, or it may be an archive of relocatable or absolute object files. For each symbol, at least the following information will be printed:

*Name*     The name of the symbol.

*Value*     Its value expressed as an offset or an address depending on its storage class.

*Size*     Its size in bytes, if available.

The output of `nm` may be controlled using the following options:

PI

`-o`     Print the value and size of a symbol in octal instead of decimal.

UN

`-x`     Print the value and size of a symbol in hexadecimal instead of decimal.

`-e`     Print only external and static symbols.

`-f`     Produce full output. Print redundant symbols (`.text`, `.data` and `.bss`), normally suppressed.

`-u`     Print undefined symbols only.

MV UN

`-V`     Print the version of the `nm` command executing on the standard error output.

## SEE ALSO

`cc(1D)`, `ld(1D)`.

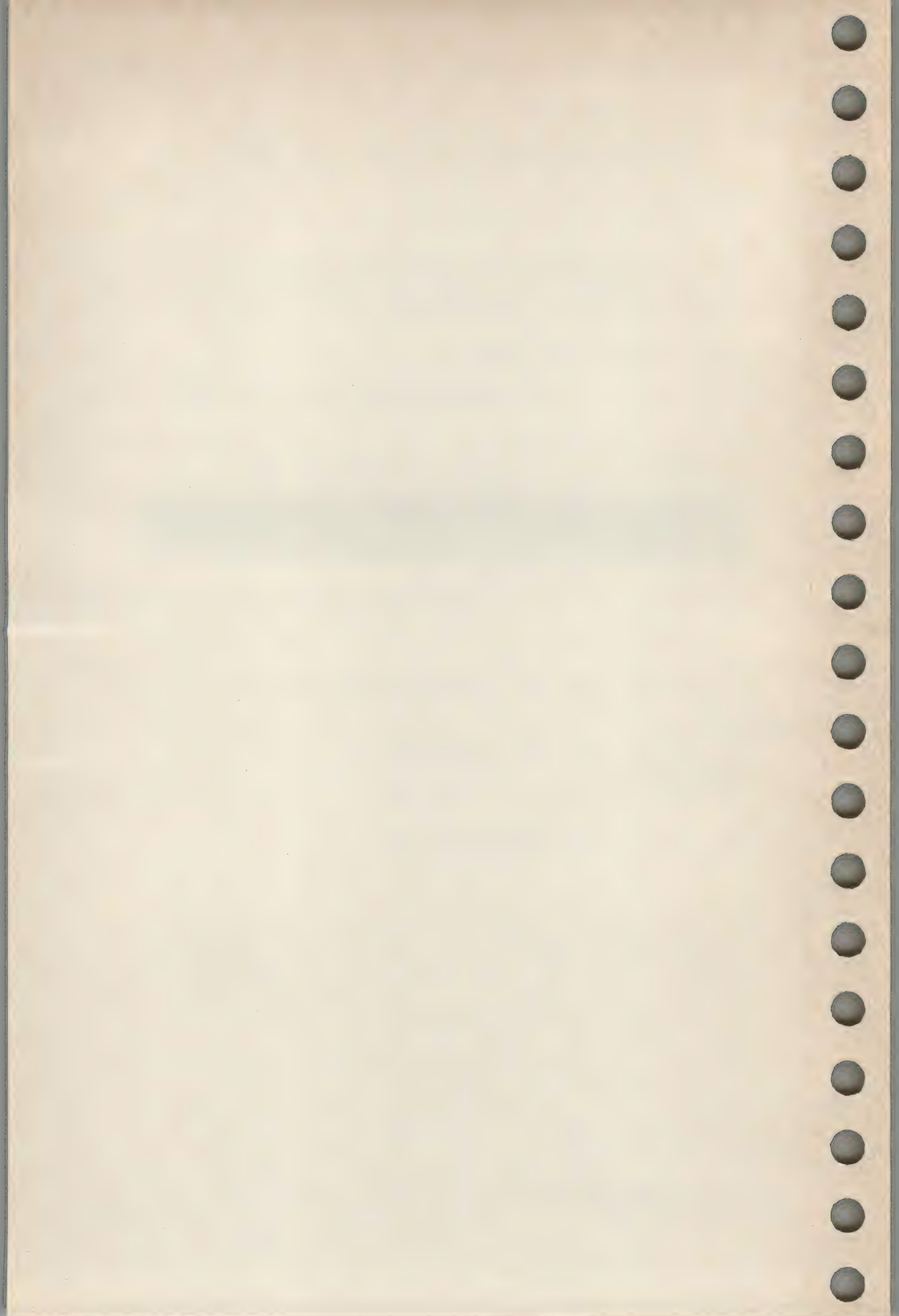
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

nohup — run a command immune to hangups and quits

## SYNOPSIS

nohup command [ arg ... ]

## DESCRIPTION

The command *nohup* executes *command* with the signals **SIGHUP** and **SIGQUIT** ignored. If output is not redirected by the user, both standard output and standard error are sent to *nohup.out*. If *nohup.out* is not writable in the current directory, output is redirected to *\$HOME/nohup.out*.

Optionally, arguments may be passed to the command by placing them as separate words after the command name.

## FILES

nohup.out          output file, if output not redirected by user

## APPLICATION USAGE

It is frequently desirable to apply *nohup* to pipelines or lists of commands. This can be done only by placing pipelines and command lists in a single file; this procedure can then be executed as *command*, and the *nohup* applies to everything in the file.

*Nohup* is typically used to prevent a program from being killed when the user logs out.

It should be noted that on some implementations of the system, logging out may destroy the environment in which the process is running. On such systems, *nohup* will not necessarily cause the process to continue after the user has logged out.

## SEE ALSO

sh(1), signal(2).

## CHANGE HISTORY

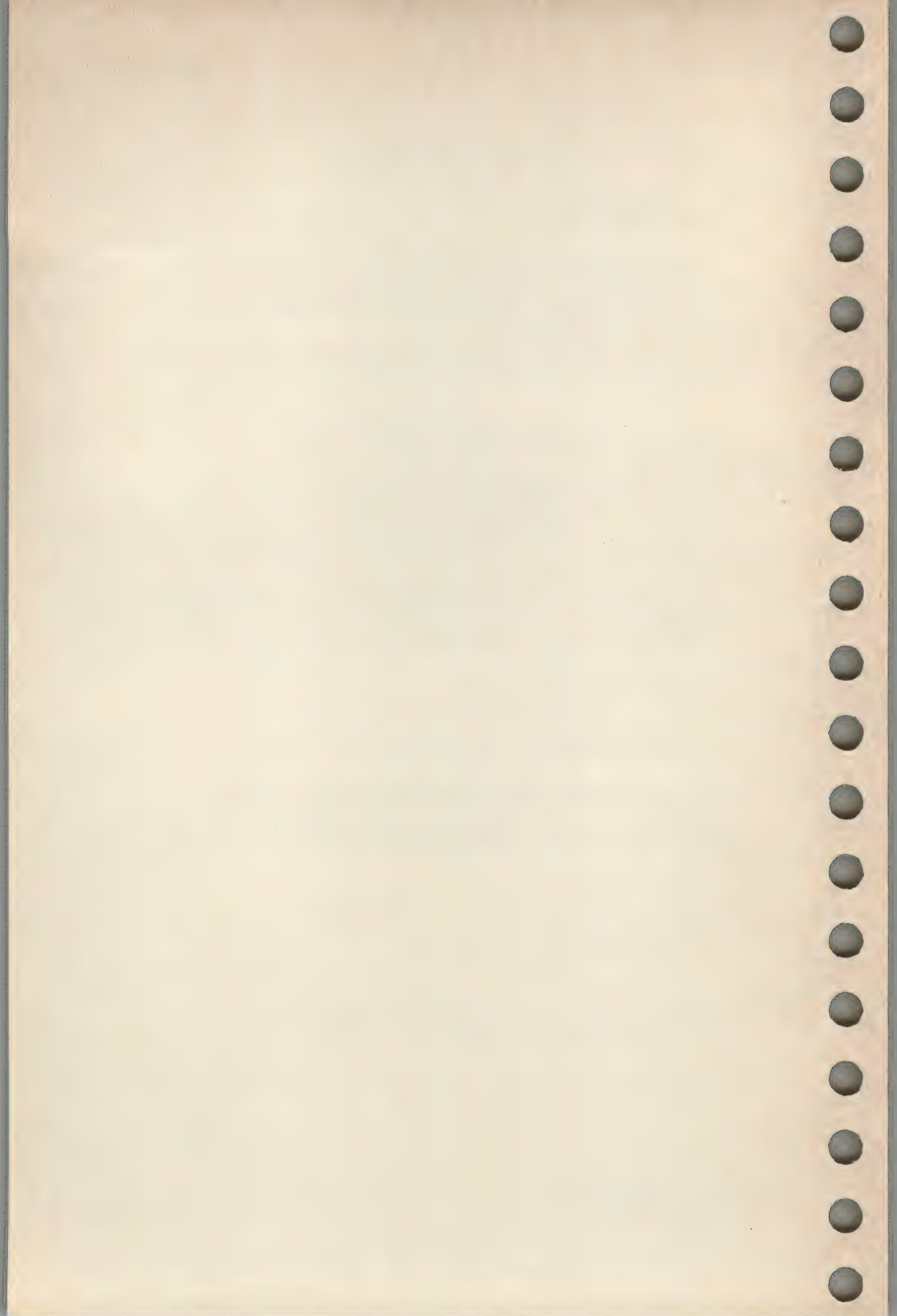
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The paragraph indicating that arguments may be passed to the command has been added.





## NAME

od — octal dump

## SYNOPSIS

od [ *—bcdosx* ] [ *file* ] [ [ *+* ] *offset* [ *.* ] [ *b* ] ]

OF

## DESCRIPTION

The command *od* writes *file* to the standard output in one or more formats as selected by the options. If no file is specified, the standard input is used. If no option is specified, the output will be in unsigned octal.

For the purposes of this description, *word* refers to a two-byte unit.

The meanings of the options are:

- b* Interpret bytes in octal.
- c* Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: NUL= \0, BS= \b, FF= \f, NL= \n, CR= \r, HT= \t; others appear as 3-digit octal numbers.
- d* Interpret *words* in unsigned decimal.
- o* Interpret *words* in unsigned octal.
- s* Interpret *words* in signed decimal.
- x* Interpret *words* in unsigned hexadecimal.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If *.* is appended, the offset is interpreted in decimal. If *b* is appended, the offset is interpreted in units of 512 bytes. If the file argument is omitted, the offset argument must be preceded by *+*.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

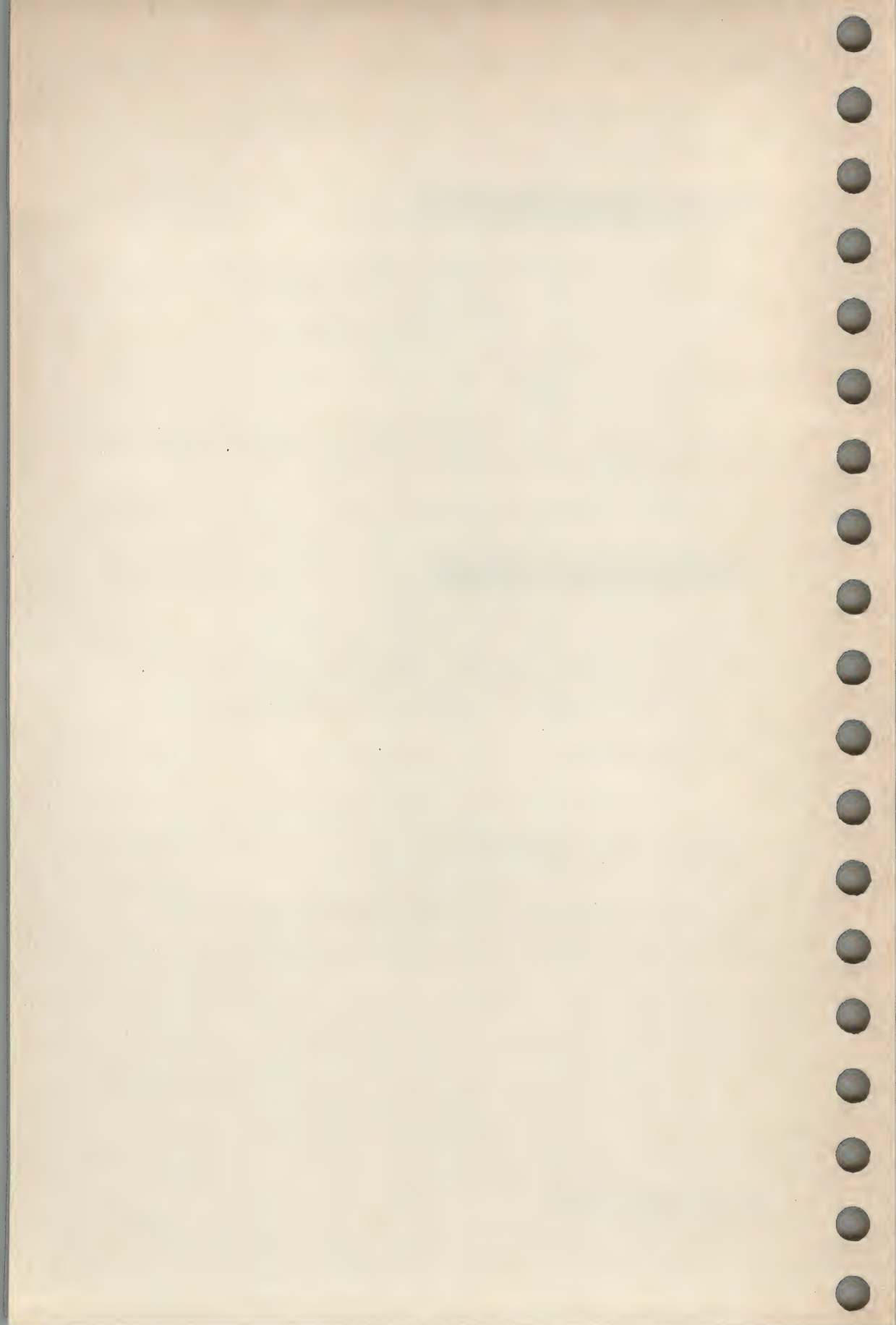
Derived from the entry in Issue 2 of the SVID with the following changes:

In the first paragraph of the **DESCRIPTION**, the words “*—o* is the default” have been replaced by “the output will be in unsigned octal”.

In the second paragraph of the **DESCRIPTION**, the words “16-bit unit, independent of the word size of the machine” have been replaced by “two-byte unit”.

The word “unsigned” has been added to the descriptions of the *—o* and *—x* options.





## NAME

pack, pcat, unpack — compress and expand files

## SYNOPSIS

pack **[-]** **[-f]** name...

pcat name ...

unpack name ...

## DESCRIPTION

## pack

The command *pack* attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file *name* is replaced by a packed file *name.z* with the same access modes, access and modified dates and owner as those of *name*. The option **-f** will force packing of *name*. This is useful for causing an entire directory to be packed even if some of the files will not benefit. If *pack* is successful, *name* will be removed. Packed files can be restored to their original form using *unpack* or *pcat*.

OF

If the **-** argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of **-** in place of *name* will cause the internal flag to be set and reset.

The command *pack* exits with an exit status that is the number of files that it failed to compress.

No packing will occur if:

- the file appears to be already packed;
- the file name has more than {NAME\_MAX}-2 characters;
- the file has links;
- the file is a directory;
- the file cannot be opened;
- the file is empty;
- no disk storage will be saved by packing;
- a file called *name.z* already exists;
- the .z file cannot be created, or
- an I/O error occurred during processing.

The last segment of the file name must contain no more than {NAME\_MAX}-2 characters to allow space for the appended .z extension.



## **pcat**

The command *pcat* does for packed files what *cat*(1) does for ordinary files, except that *pcat* cannot be used as a filter. The specified files are unpacked and written to the standard output. Thus to view a packed file named *name.z* use:

```
pcat name.z
```

or

```
pcat name
```

To make an unpacked copy, called *abc*, of a packed file named *name.z* (without destroying *name.z*) use the command:

```
pcat name >abc
```

The command *pcat* returns the number of files it was unable to unpack. Failure may occur if:

- the file name (exclusive of the *.z*) has more than {*NAME\_MAX*}-2 characters;
- the file cannot be opened, or
- the file does not appear to be the output of *pack*.

## **unpack**

The command *unpack* expands files created by *pack*. For each file *name* specified in the command, a search is made for a file called *name.z* (or just *name*, if *name* ends in *.z*). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the *.z* suffix stripped from its name, and has the same access modes, access and modification dates and owner as those of the packed file.

The command *unpack* exits with an exit status that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in *pcat*, as well as for the following:

- a file with the "unpacked" name already exists, or
- the unpacked file cannot be created.

## **SEE ALSO**

*cat*(1).

## **APPLICATION USAGE**

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree may form the first part of each *.z* file, it is usually not worthwhile to pack small files, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Object files, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

Packed files are not necessarily portable to other systems.

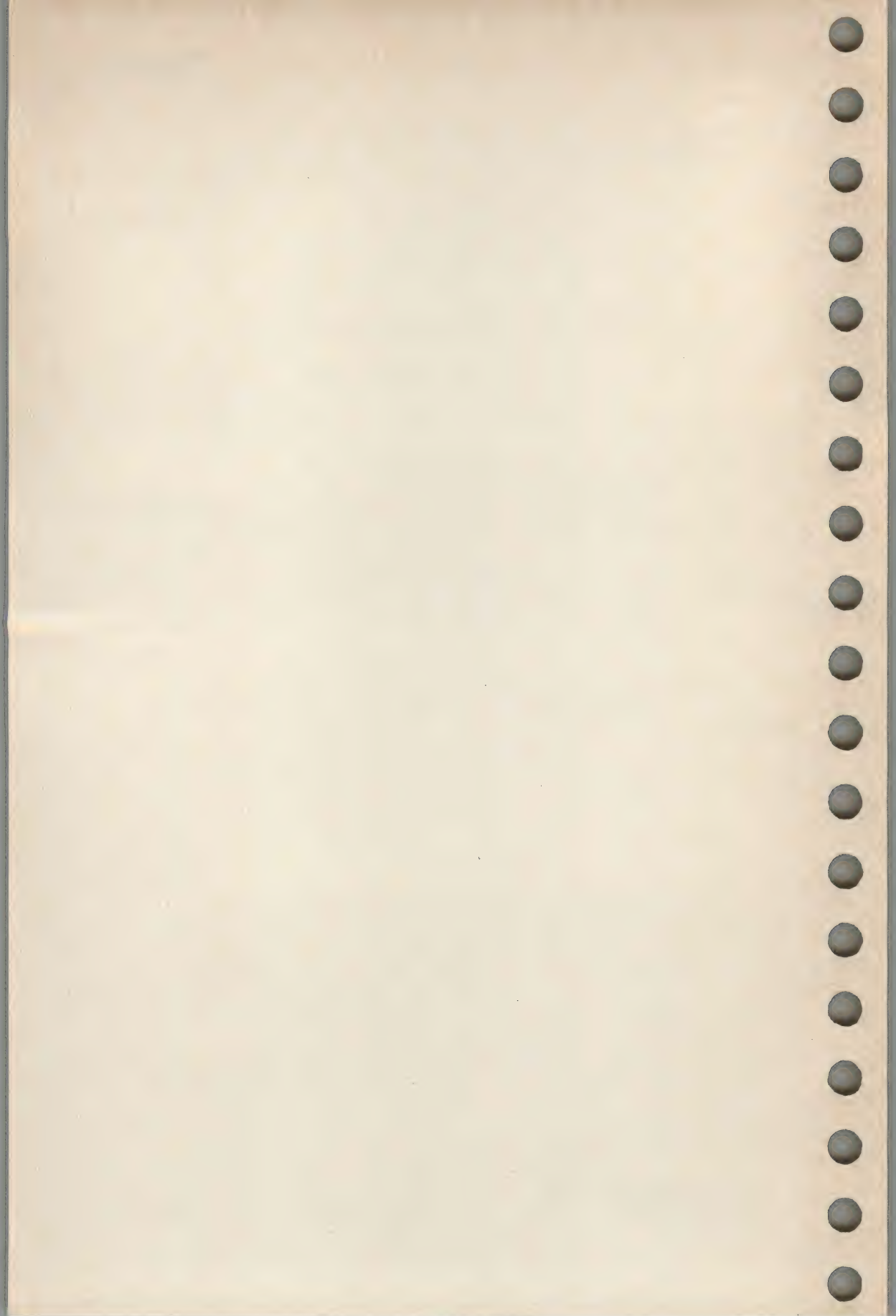
**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID.





## NAME

`passwd` — change login password

## SYNOPSIS

`passwd [name]`

UN

## DESCRIPTION

The command `passwd` changes or installs a password associated with the login name.

Users may change only the password which corresponds to their login *name*.

The command `passwd` prompts users for their old password, if any. It then prompts for the new password twice. If password aging is in effect, then the first time the new password is entered, `passwd` checks to see if the old password has aged sufficiently. If aging is insufficient the new password is rejected and `passwd` terminates.

If aging is sufficient, a check is made to ensure that the new password meets construction requirements. When the new password is entered a second time, the two copies of the new password are compared. If the two copies are not identical the cycle of prompting for the new password is repeated for at most two more times.

The super-user may change any password: hence `passwd` does not prompt the super-user for the old password. The super-user is not forced to comply with password aging and password construction requirements. The super-user can create a null password by entering a carriage return in response to the prompt for a new password.

## FILES

`/etc/passwd`      system's password file

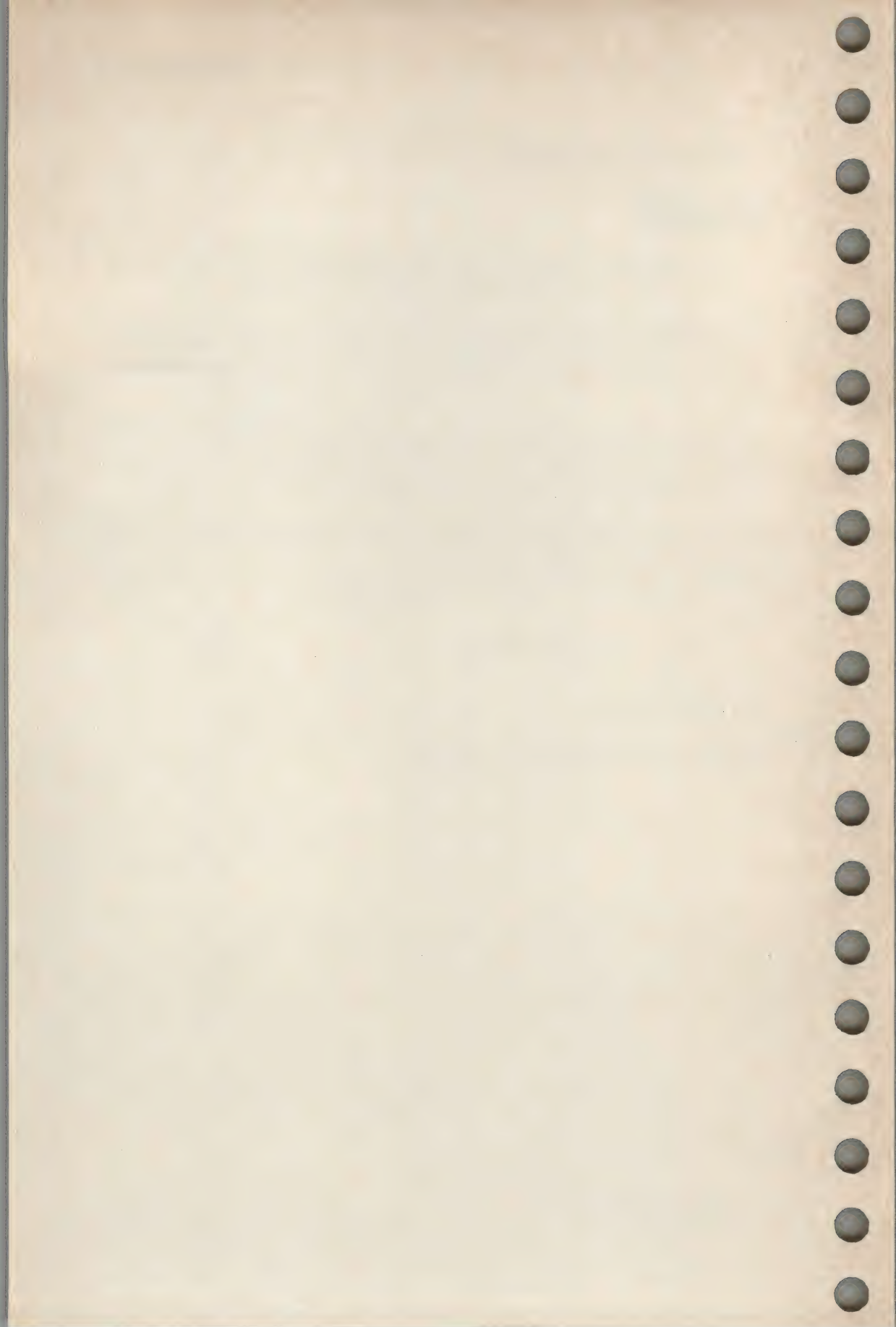
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

paste — merge same lines of several files or subsequent lines of one file

## SYNOPSIS

```
paste file ...
paste -d list file ...
paste -s [-d list] file ...
```

## DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files. The file name — means standard input. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). In the last form above (—s option), *paste* combines subsequent lines of the input file (serial merging).

In all cases, lines are glued together with the tab character, unless the —d option is used (see below).

Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if — is used in place of a file name.

The meanings of the options are:

- d Without this option, the newline characters of each but the last file (or last line in case of the —s option) are replaced by a tab character. This option allows replacing the tab character by one or more alternate characters (see below).
- list* One or more characters immediately following —d replace the default tab as the line concatenation character. The list is used circularly, i.e., when exhausted, it is reused. In parallel merging (i.e., no —s option) the lines from the last file are always terminated with a newline character, not from the *list*. The list may contain the special escape sequences: \n (newline), \t (tab), \\ (backslash), and \0 (empty string, not a null character). Quoting may be necessary if characters have special meaning to the command interpreter.
- s Merge subsequent lines rather than one from each input file. Use tab for concatenation, unless a *list* is specified with the —d option. Regardless of the *list*, the very last character of the file is forced to be a newline.

## EXAMPLES

1. The following example lists the contents of the current directory in four columns

```
ls | paste — — — —
```

2. The following example combines pairs of lines into lines

```
paste -s -d "\t\n" file
```

## SEE ALSO

cut(1), grep(1), pr(1).

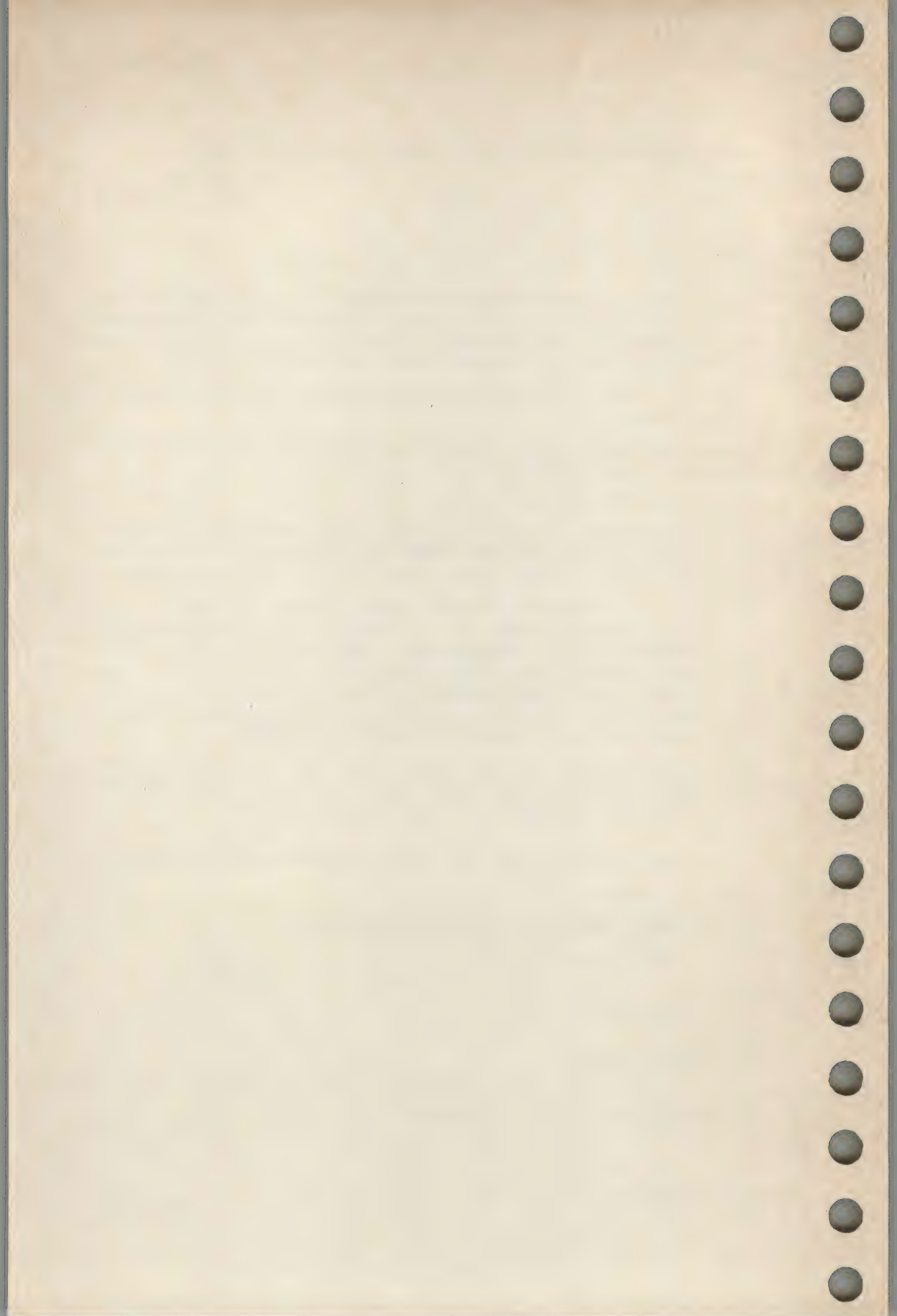
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`pg` — file perusal filter for soft-copy terminals

## SYNOPSIS

`pg` [`—number`] [`—p string`] [`—cefmts`] [`+linenumber`] [`+ /re/`] [`file...`]

## DESCRIPTION

The command `pg` is a filter that allows the examination of the named *file* or files one screenful at a time on a soft-copy terminal. (The file name — and/or null arguments indicate that `pg` should read from the standard input.) Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are enumerated below.

This command is different from previous paginators in that it allows the user to back up and review something that has already passed. The method for doing this is explained below.

In order to determine terminal attributes, `pg` may optionally attempt to search a system-dependent database, keyed on the value of the `TERM` environment variable. If no information is available, a dumb terminal is assumed.

The command line options are:

`—number`

An integer specifying the size (in lines) of the window that `pg` is to use instead of the default. (On a terminal containing 24 lines, the default window size is 23.)

`—p string`

Causes `pg` to use *string* as the prompt. If the prompt string contains a `%d`, the first occurrence of `%d` in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is .

`—c` Home the cursor and clear the screen before displaying each page. This option is ignored if the terminal does not support this.

`—e` Causes `pg` not to pause at the end of each file.

`—f` Normally, `pg` splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results. The `—f` option inhibits `pg` from splitting lines.

UN

`—n` Normally, commands must be terminated by a newline character. This option causes an automatic end of command as soon as a command letter is entered.

`—s` Causes `pg` to print all messages and prompts in standout mode (usually inverse video) if the terminal supports this.

`+ linenumber`

Start up at *linenumber*.

`+ /re/` Start up at the first line containing the regular expression *re*.



The responses that may be typed when *pg* pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding address; an optionally signed number indicating the point from which further text should be displayed. This address is interpreted in either pages or lines depending on the command. A signed address specifies a point relative to the current page or line, and an unsigned address specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are as follows:

**(+ 1)newline or space**

This causes one page to be displayed. The address is specified in pages.

**(+ 1)l**

With a relative address this causes *pg* to simulate scrolling the screen, forward or backward, the number of lines specified. With an absolute address this command prints a screenful beginning at the specified line.

**(+ 1)d or ^D**

Simulates scrolling half a screen forward or backward.

The following perusal commands take no address.

**. or ^L**

Typing a single period causes the current page of text to be redisplayed.

**\$** Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available for searching for text patterns in the text. Simple regular expression syntax is available. The *res* must always be terminated by a *<newline>*, even if the *—n* option is specified.

*i/re/* Search forward for the *i*th (default *i*=1) occurrence of *re*. Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

**^re**

*?re?* Search backwards for the *i*th (default *i*=1) occurrence of *re*. Searching begins immediately before the current page and continues to the beginning of the current file, without wrap-around. (The *^* notation is useful for terminals that do not properly handle the *?*.)

After searching, *pg* will normally display the line found at the top of the screen. This can be modified by appending *m* or *b* to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix *t* can be used to restore the original situation.

The user of *pg* can modify the environment of perusal with the following commands:

*in* Begin perusing the *i*th next file in the command line. The *i* is an unsigned number; default value is 1.

*ip* Begin perusing the *i*th previous file in the command line. *i* is an unsigned number; default is 1.

*iw* Display another window of text. If *i* is present, set the window size to *i*.

*s filename*

Save the input in the file named *filename*. Only the current file being perused is saved. The white space between the *s* and *filename* is optional. This command must always be terminated by a <newline>, even if the *—n* option is specified.

*h* Help by displaying an abbreviated summary of available commands.

*q* or *Q*

Quit *pg*.

*!command*

The argument *command* is passed to the command interpreter, whose name is taken from the SHELL environment variable. If this is not available, the default command interpreter is used. This command must always be terminated by a <newline>, even if the *—n* option is specified.

At any time when output is being sent to the terminal, receipt of a quit or interrupt signal causes *pg* to stop sending output and to display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the signal occurs.

If the standard output is not a terminal, *pg* acts just like the *cat* command, except that a header is printed before each file (if there is more than one).

#### SEE ALSO

*grep*(1).

#### APPLICATION USAGE

While waiting for terminal input, *pg* responds to SIGINT and SIGQUIT signals by terminating execution. Between prompts, however, these signals interrupt *pg*'s current task and place the user in prompt mode. These signals should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

If terminal tabs are not set every eight positions, undesirable results may occur.

When *pg* is used as a filter with another command that changes the terminal I/O options, terminal settings may not be restored correctly.

The *—n* option relies on the *termio*(7) terminal interface. If *pg* is being used over an alternative interface, the *—n* option may not be supplied.



**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID with the following change:

The words "if the terminal supports this" have been added to the description of the —s option.

## NAME

pr — print files

## SYNOPSIS

pr [options] [file...]

## DESCRIPTION

The command *pr* prints *file* or files on the standard output. If *file* is —, or if no files are specified, the standard input is assumed. By default, the listing is separated into pages, the size of the page being system dependent, each headed by the page number, a date and time, and the name of the file.

By default, columns are of equal width, separated by at least one space; lines which do not fit are truncated. If the —s option is used, lines are not truncated and columns are separated by the separation character.

If the standard output is associated with a terminal, error messages are withheld until *pr* has completed printing.

The options below may appear singly, or may be combined in any order:

PI **+k** Begin printing with page *k* (default is 1).

—*k* Produce *k*-column output (default is 1). This option should not be used with —*m*. The options —*e* and —*i* are assumed for multi-column output.

—*a* Print multi-column output across the page. This option is appropriate only with the —*k* option.

—*m* Merge and print all files simultaneously, one per column (overrides the —*k* option).

—*d* Double-space the output.

—*eck*

Expand *input* tabs to character positions *k*+1, 2•*k*+1, 3•*k*+1, etc.. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If *c* (any non-digit character) is given, it is treated as the input tab character (default for *c* is the tab character).

—*ick* In *output*, replace white space wherever possible by inserting tabs to character positions *k*+1, 2•*k*+1, 3•*k*+1, etc.. If *k* is 0 or is omitted, default tab settings at every eighth position are assumed. If *c* (any non-digit character) is given, it is treated as the output tab character (default for *c* is the tab character).

—*nck* Provide *k*-digit line numbering (default for *k* is 5). The number occupies the first *k*+1 character positions of each column of normal output or each line of —*m* output. If *c* (any non-digit character) is given, it is appended to the line number to separate it from whatever follows (default for *c* is a tab).

—*wk* Set the width of a line to *k* character positions for multi-column output (default is 72 for equal-width, multi-column output; no limit otherwise).



- ok Offset each line by *k* character positions (default is 0). The number of character positions per line is the sum of the width and offset.
- lk Set the length of a page to *k* lines (default is system dependent). If *k* is less than what is needed for the page header and trailer, then the option —*t* is in effect; that is, header and trailer lines are suppressed in order to make room for text.
- h *header*  
Use *header* as the header to be printed instead of the file name.
- p Pause before beginning each page if the output is directed to a terminal (*pr* will ring the bell at the terminal and wait for a carriage return).
- f Use form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.
- r Print no diagnostic reports on failure to open files.
- t Print neither the five-line identifying header nor the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.
- sc Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

#### EXAMPLES

1. Print *file1* and *file2* as a double-spaced, three-column listing headed by *file list*:  

```
pr -3dh "file list" file1 file2
```
2. Write *file1* on *file2*, expanding tabs to columns 10, 19, 28, ... :  

```
pr -e9 -t <file1 >file2
```

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

Words have been added to the first paragraph and the description of the —*l* option indicating that the default page length is system dependent.

The default for —*w* has had the text "for equal-width, multi-column output; no limit otherwise" added from the System V Release 2.0 manual.

## NAME

`prof` — display profile data (OPTIONAL)

## SYNOPSIS

`prof [ -tcan ] [ -ox ] [ -g ] [ -z ] [ -m mdata ] [ prog ]`

PI

## DESCRIPTION

The command *prof* interprets a *profile file* produced by the *monitor* routine. The symbol table in the object file *prog* (*a.out* by default) is read and correlated with a profile file (*mon.out* by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options *t*, *c*, *a*, and *n* determine the type of sorting of the output lines:

- t* Sort by decreasing percentage of total time (default).
- c* Sort by decreasing number of calls.
- a* Sort by increasing symbol address.
- n* Sort lexically by symbol name.

The mutually exclusive options —*o* and —*x* specify the printing of the address of each symbol monitored:

- o* Print each symbol address (in octal) along with the symbol name.
- x* Print each symbol address (in hexadecimal) along with the symbol name.

The following options may be used in any combination:

- g* Include non-global symbols (static functions).
- z* Include all symbols in the profile range, even if associated with zero number of calls and zero time.

—*m mdata*

Use file *mdata* instead of *mon.out* as the input profile file.

A program creates a profile file if it has been loaded with the —*p* option of *cc*. This option to the *cc* command arranges for calls to *monitor* at the beginning and end of execution. It is the call to *monitor* at the end of execution that causes a profile file to be written. The number of calls to a function is tallied if the —*p* option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environment variable PROFDIR. If PROFDIR is not set, *mon.out* is produced in the directory current when the program terminates. If PROFDIR = *string*, *string/pid.progname* is produced, where *progname* consists of *argv[0]* with any path prefix removed, and *pid* is the program's process ID. If PROFDIR is set, but null, no profiling output is produced.



## FILES

mon.out	for profile
a.out	for namelist

## SEE ALSO

cc(1D), exit(2), profil(2), monitor(3C).

## APPLICATION USAGE

The times reported in successive identical runs may show variances, because of varying cache-hit ratios due to sharing of the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data.

In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call *exit* or return from *main* are guaranteed to produce a profile file, unless a final call to monitor is explicitly coded.

Optional. Requires the *profil* system service routine.

## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

References to the SVID entry MARK(SD\_LIB) have been removed.

## NAME

*prs* — print an SCCS file

## SYNOPSIS

*prs* [ options ] file ...

## DESCRIPTION

The command *prs* prints, on the standard output, parts or all of an SCCS file in a user-supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with *s.*) and unreadable files are silently ignored. If a name of — is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of options and file names.

All the described options apply independently to each named file.

—d[*dataspec*]

Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *Data Keywords*) interspersed with optional user supplied text.

—r[*SID*]

Used to specify the SCCS identification string (*SID*) of a delta for which information is desired. If no *SID* is specified, the *SID* of the most recently created delta is assumed.

—e Requests information for all deltas created earlier than and including the delta designated via the —r option or the date given by the —c option.

—l Requests information for all deltas created later than and including the delta designated via the —r option or the date given by the —c option.

UN

—c[*date-time*]

The cutoff *date-time* is in the form:

YY[MM[DD[HH[MM[SS]]]]]

Units omitted from the date-time default to their maximum possible values; for example, —c7502 is equivalent to —c750228235959. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date in the form: —c77/2/2

—a Requests printing of information for both removed, i.e., delta type = *R*, see *rmDEL(1D)* and existing, i.e., delta type = *D*, deltas. If the —a option is not specified, information for existing deltas only is provided.



## Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognised data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either Simple (S), in which keyword substitution is direct, or Multi-line (M), in which keyword substitution is followed by a <carriage return>.

User-supplied text is any text other than recognised data keywords. A tab is specified by \t and <carriage return>/ <newline> is specified by \n. The default data keywords are:

:Dt: \t:DL: \nMRs: \nMR:COMMENTS: \n:C:

TABLE 1. SCCS Files Data Keywords

Keyword	Data Item	File Section	Value	Format
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	D or R	S
:I:	SCCS ID string (SID)	"	:R:/:L:/:B:/:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:/:Tm:/:Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS:/:DS:...	S

:Dx:	Deltas excluded (seq #)	"	:DS: :DS: ...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS: ...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	yes or no	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	yes or no	S
:KV:	Keyword validation string	"	text	S
:BF:	Branch flag	"	yes or no	S
:J:	Joint edit flag	"	yes or no	S
:LK:	Locked releases	"	:R: ...	S
:Q:	User-defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	yes or no	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of <i>what</i> (1D) string	N/A	:Z::M: \t:l:	S
:A:	A form of <i>what</i> (1D) string	N/A	:Z::Y: :M: :l::Z:	S
:Z:	<i>what</i> (1D) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

\* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

## EXAMPLES

1. The following example:

```
prs -d"User Names for :F: are: \n:UN:" s.file
```

may produce on the standard output:

User Names for s.file are:

xyz

131

abc



2. The following example:

```
prs -d"Delta for pgm :M:: :l: - :D: By :P:" -r s.file
```

may produce on the standard output:

```
Delta for pgm main.c: 3.7 - 77/12/1 By cas
```

3. As a special case:

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
```

```
MRs:
```

```
bl78-12345
```

```
bl79-54321
```

```
COMMENTS:
```

```
this is the comment line for s.file initial delta
```

for each delta table entry of the *D* type. The only option allowed to be used with the special case is the *-a* option.

## SEE ALSO

admin(1D), delta(1D), get(1D), what(1D).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The descriptions of the *-d* and *-r* options have been corrected from the System V Release 2.0 manual page.

The description of the *-e* option has been added.

## NAME

ps — report process status

## SYNOPSIS

ps [ options ]

OF PI UN

## DESCRIPTION

The command *ps* prints certain information about active processes. Without *options*, information is printed about processes associated with the current terminal. The output consists of a short listing containing only the process ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the selection of *options*.

The *options* using lists as arguments can have the list specified in one of two forms: a list of identifiers separated from one another by a comma, or a list of identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

The *options* are:

- e Print information about all processes.
- d Print information about all processes, except process group leaders.
- a Print information about all processes, except process group leaders and processes not associated with a terminal.
- f Generate a full listing. (See below for meaning of columns in a full listing.)
- l Generate a long listing. (See below.)
- n *namelist*  
The argument will be taken as the name of an alternate system *namelist* file in place of the default.
- t *termlist*  
Restrict listing to data about the processes associated with the terminals given in *termlist*. Terminal identifiers may be specified in one of two forms: the device's file name (e.g., *tty04*) or if the device's file name starts with *tty*, just the digit identifier (e.g., *04*).
- p *proclist*  
Restrict listing to data about processes whose process ID numbers are given in *proclist*.
- u *uidlist*  
Restrict listing to data about processes whose user ID numbers or login names are given in *uidlist*. In the listing, the numerical user ID will be printed unless the —f option is used, in which case the login name will be printed.
- g *grplist*  
Restrict listing to data about processes whose process group leaders are given in *grplist*.



The column headings and the meaning of the columns in a *ps* listing are given below; the letters *f* and *l* indicate the option (*full* or *long*) that causes the corresponding heading to appear; *all* means that the heading always appears. Note that these two options determine only what information is provided for a process; they do not determine which processes will be listed.

F	(l)	Flags (octal and additive) associated with the process.
S	(l)	The state of the process.
UID	(f,l)	The user ID number of the process owner; the login name is printed under the — <i>f</i> option.
PID	(all)	The process ID of the process; it is possible to kill a process if this datum is known.
PPID	(f,l)	The process ID of the parent process.
C	(f,l)	Processor utilisation for scheduling.
PRI	(l)	The priority of the process; higher numbers mean lower priority.
NI	(l)	Nice value; used in priority computation.
ADDR	(l)	The memory address of the process.
SZ	(l)	The size in blocks of the core image of the process.
WCHAN	(l)	The event for which the process is waiting or sleeping; if blank, the process is running.
STIME	(f)	Starting time of the process.
TTY	(all)	The controlling terminal for the process.
TIME	(all)	The cumulative execution time for the process.
CMD	(all)	The command name; the full command name and its arguments are printed under the — <i>f</i> option.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked *defunct*.

Under the option —*f*, *ps* tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the option —*f*, is printed in square brackets.

#### FILES

/etc/passwd supplies UID information

SEE ALSO

kill(1), nice(1).

APPLICATION USAGE

Things can change while *ps* is running; the snap-shot it gives is only true for an instant, and may not be accurate by the time it is displayed.

On some systems, some of the information described here may not be available, and will consequently not be presented.

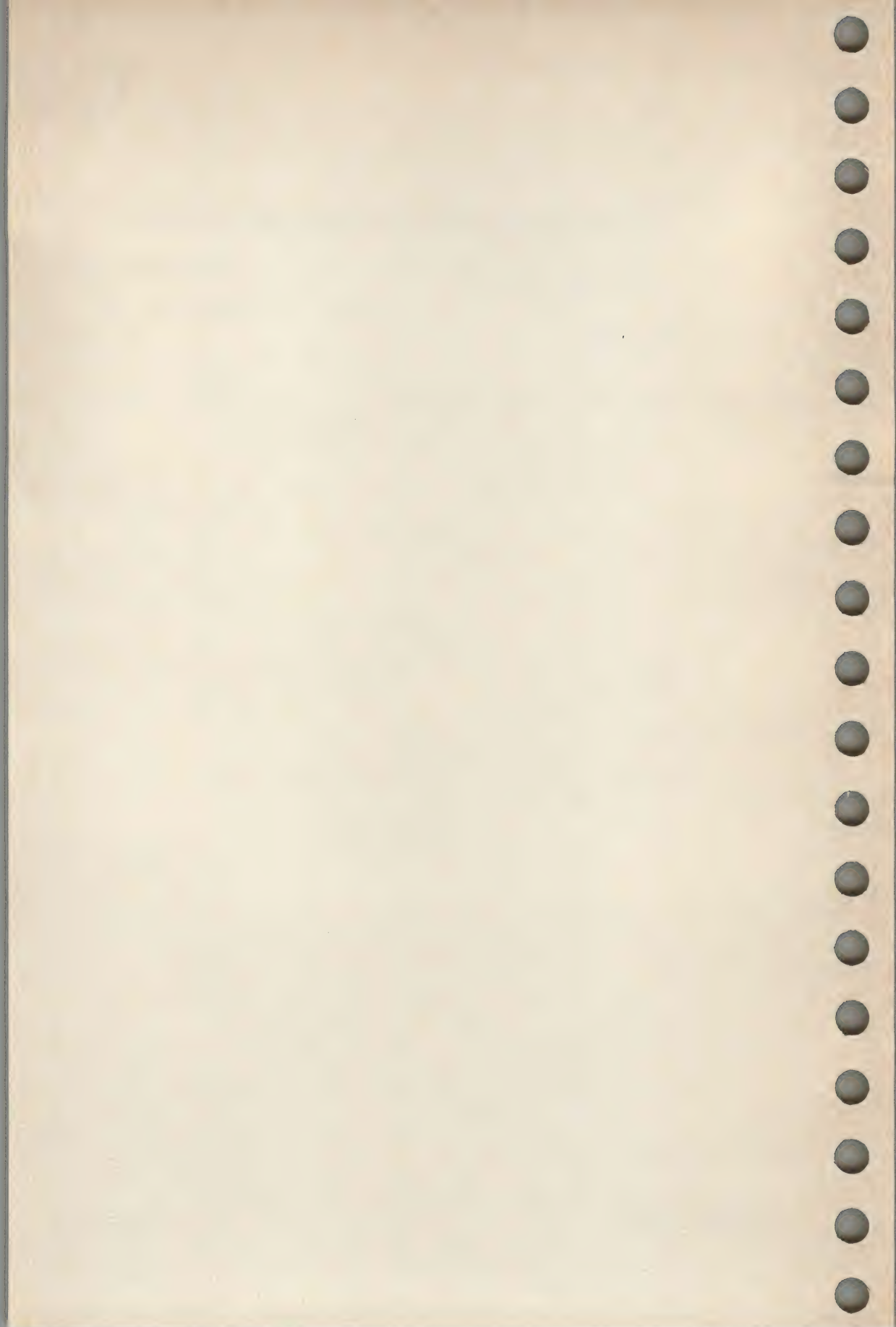
CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID.





NAME

`pwd` — print working directory name

SYNOPSIS

`pwd`

DESCRIPTION

The command *pwd* writes the path name of the working (current) directory to the standard output.

SEE ALSO

`cd(1)`, `getcwd(3C)`.

APPLICATION USAGE

This is usually a shell built-in command.

CHANGE HISTORY

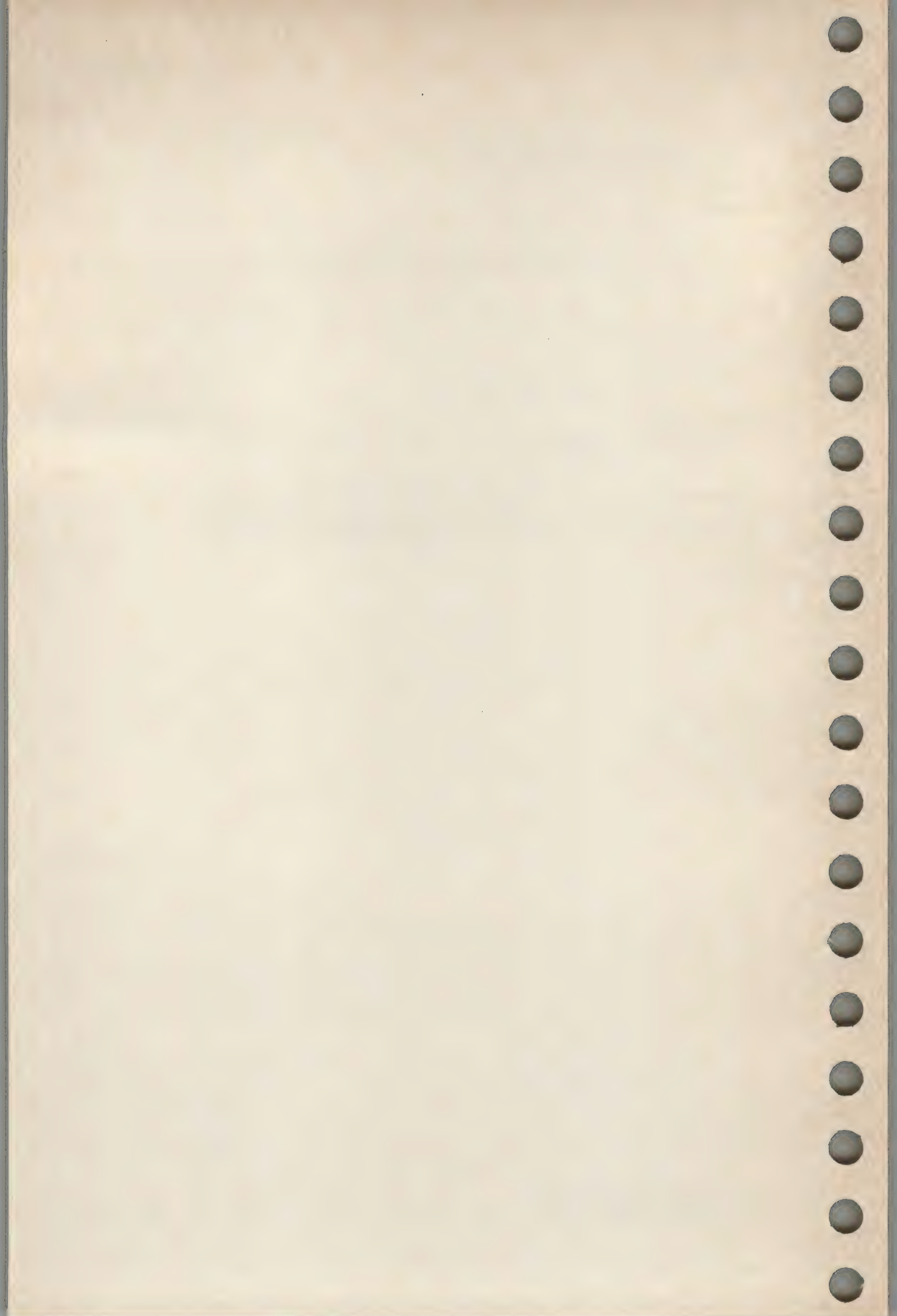
First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The words "to the standard output" have been added to the description.





## NAME

`rm`, `rmdir` — remove files or directories

## SYNOPSIS

`rm [ —fri ] file ...`

`rmdir dir ...`

## DESCRIPTION

**rm** The command *rm* removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with *y* the file is deleted, otherwise the file remains. No questions are asked when the option *—f* is given or if the standard input is not a terminal.

If a designated file is a directory, an error comment is printed unless the optional argument *—r* has been used. In that case, *rm* recursively deletes the entire contents of the specified directory and the directory itself.

If the option *—i* (interactive) is in effect, *rm* asks whether to delete each file and, under *—r*, whether to examine each directory.

**rmdir**

The command *rmdir* removes entries for the named directories, which must be empty.

## APPLICATION USAGE

It is forbidden to remove the file `..` in order to avoid the consequences of inadvertently doing something like:

```
rm —r .*
```

## SEE ALSO

`unlink(2)`.

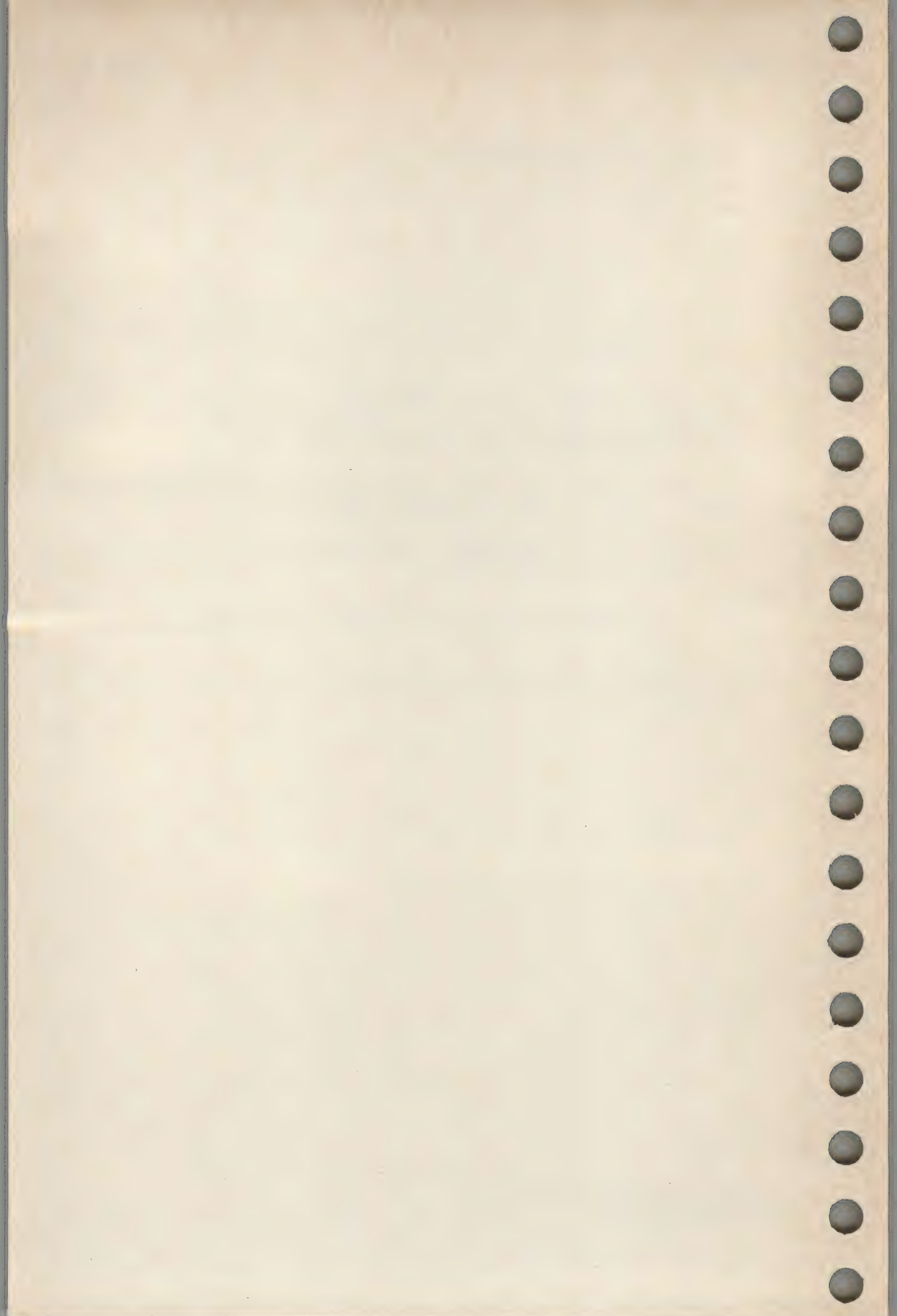
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

`rm del` — remove a delta from an SCCS file

## SYNOPSIS

`rm del` —rSID file...

## DESCRIPTION

The command `rm del` removes the delta specified by the SID from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must not be that of a version being edited for the purpose of making a delta (i.e., if a *p-file*, see `get(1D)`, exists for the named SCCS file, the SID specified must not appear in any entry of the *p-file*).

If a directory is named, `rm del` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with *s.*) and unreadable files are silently ignored. If a name of — is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The restrictions on removal of a delta are: (1) the user who made a delta can remove it; (2) the owner of the file and directory can remove a delta.

## SEE ALSO

`delta(1D)`, `get(1D)`, `prs(1D)`.

## CHANGE HISTORY

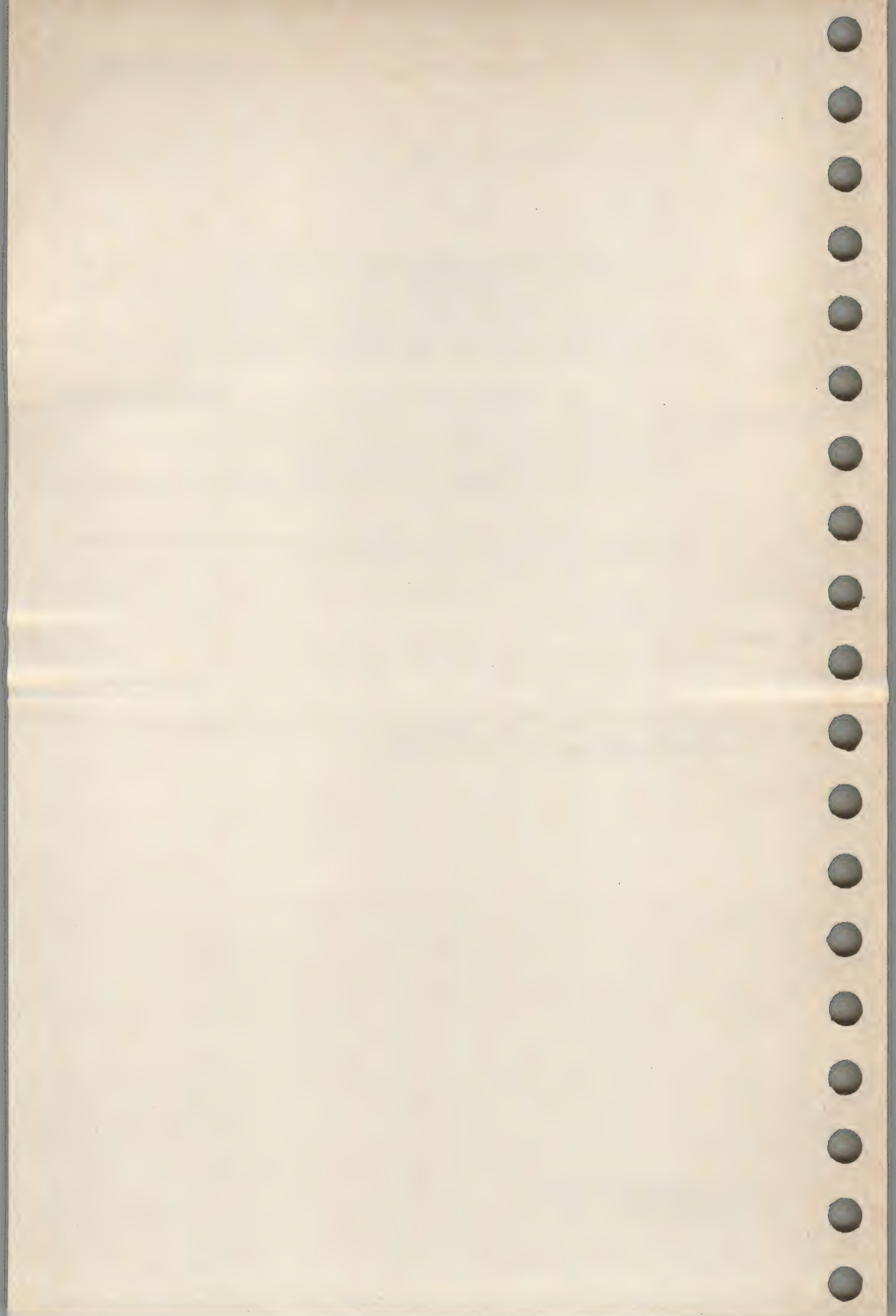
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The SYNOPSIS has been corrected from "`rm del —r`".





## NAME

sact — print current SCCS file editing activity

## SYNOPSIS

sact file...

## DESCRIPTION

The command *sact* informs the user of any impending deltas to a named SCCS file. This situation occurs when *get —e* has been previously executed without a subsequent execution of *delta*. If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of — is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1 specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

Field 2 specifies the SID for the new delta to be created.

Field 3 contains the logname of the user who will make the delta (i.e., executed a *get* for editing).

Field 4 contains the date that *get —e* was executed.

Field 5 contains the time that *get —e* was executed.

## SEE ALSO

delta(1D), get(1D), unget(1D).

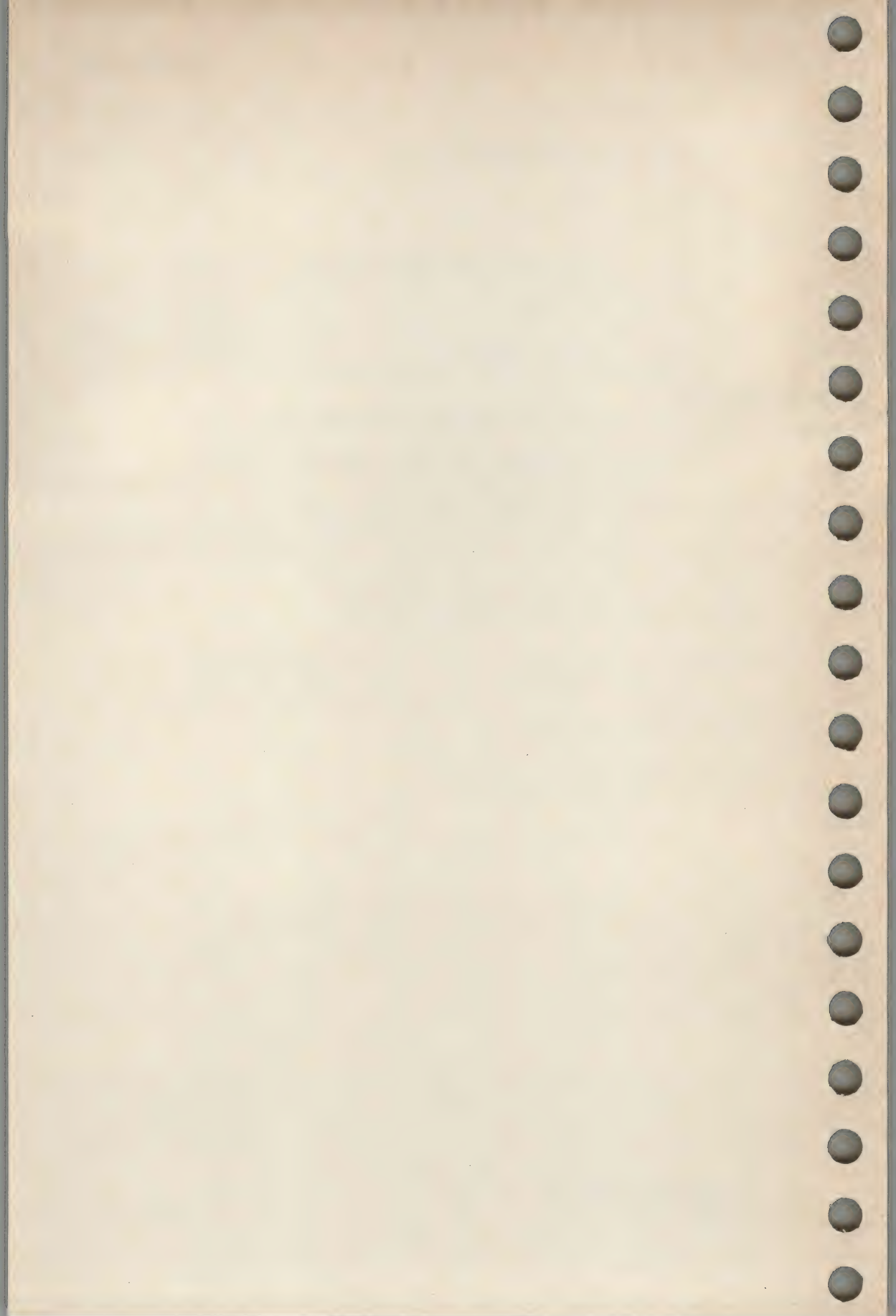
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

sdb — symbolic debugger (OPTIONAL)

## SYNOPSIS

```
sdb [ obfile [ corfile [ directory-list ] ] ]
```

OP UN

## DESCRIPTION

The command *sdb* is a symbolic debugger that can be used with C, and possibly programs in other languages. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

The argument *obfile* is an executable program file which has been compiled with the *—g* (debug) option; if it has not been compiled with the *—g* option, or if it is not an executable file, the symbolic capabilities of *sdb* will be limited, but the file can still be examined and the program debugged. The default for *obfile* is *a.out*. The argument *corfile* is assumed to be a core image file produced after executing *obfile*; the default for *corfile* is *core*. The core file need not be present. A *—* in place of *corfile* will force *sdb* to ignore any core image file. The colon-separated list of directories (*directory-list*) is used to locate the source files used to build *obfile*.

It is useful to know that at any time there is a *current line* and *current file*. If *corfile* exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in *main()*. The current line and file may be changed with the source file examination commands.

By default, warnings are provided if the source files used in producing *obfile* cannot be found, or are newer than *obfile*.

Names of variables are written just as they are in C. (The command *sdb* does not truncate names.) Variables local to a procedure may be accessed using the form *procedure:variable*. If no procedure name is given, the procedure containing the current line is used by default.

It is also possible to refer to structure members as *variable.member*, pointers to structure members as *pointer->member* and array elements as *variable[number]*. Pointers may be dereferenced by using the form *pointer[0]*. Combinations of these forms may also be used. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure referenced by *sdb*. An unqualified structure variable may also be used with various commands. Generally, *sdb* will interpret a structure as a set of variables. Thus, *sdb* will display the values of all the elements of a structure when it is requested to display a structure. An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value *sdb* displays, not the addresses of individual elements.

Elements of a multidimensional array may be referenced as *variable[number][number]...*, or as *variable[number,number,...]*. In place of *number*, the form *number;number* may be used to indicate a range of values, *\** may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. As with structures,



*sdb* displays all the values of an array or of the section of an array if trailing subscripts are omitted. It displays only the address of the array itself or of the section specified by the user if subscripts are omitted.

A particular instance of a variable on the stack may be referenced by using the form *procedure:variable,number*. All the variations mentioned in naming variables may be used. *Number* is the occurrence of the specified procedure on the stack, counting the top, or most current, as the first. If no procedure is specified, the procedure currently executing is used by default.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as *file-name:number* or *procedure:number*. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under *sdb*, all addresses refer to the executing program.

#### Commands

The commands for examining data in the program are:

- t Print a stack trace of the terminated or halted program.
- T Print the top line of the stack trace.

#### *variable/clm*

Print the value of *variable* according to length *l* and format *m*. A numeric count *c* indicates that a region of memory, beginning at the address implied by *variable*, is to be displayed. The length specifiers are:

- b one byte
- h two bytes (half word)
- l four bytes (long word)

Legal values for *m* are:

- c character
- d decimal
- u decimal, unsigned
- o octal
- x hexadecimal
- s assume *variable* is a string pointer and print characters starting at the address pointed to by the variable.
- a print characters starting at the variable's address. This format may not be used with register variables.
- p pointer to procedure
- i disassemble machine-language instruction with addresses printed numerically and symbolically.

The length specifiers are only effective with the formats *c*, *d*, *u*, *o* and *x*. Any of the specifiers, *c*, *l*, and *m*, may be omitted. If all are omitted, *sdb* chooses a length and a format suitable for the variable's type as declared in the program. If *m* is specified, then this format is used for displaying the variable. A length specifier determines the output length of the value to be displayed, sometimes resulting in truncation. A count specifier *c* tells *sdb* to display that many units of memory, beginning at the address of *variable*. The number of bytes in one such unit of memory is determined by the length specifier *l*, or if no length is given, by the size associated with the *variable*. If a count specifier is used for the *s* or *a* command, then that many characters are printed. Otherwise successive characters are printed until either a null byte is reached or 128 characters are printed. The last variable may be redisplayed with the command *./*.

*linenumber?lm*

*variable:?lm*

Print the value at the address from *a.out* or 1 space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is *i*.

*variable=lm*

*linenumber=lm*

*number=lm*

Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then *lx* is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

*variable!value*

Set *variable* to the given *value*. The *value* may be a number, a character constant or a variable. The *value* must be well defined; expressions which produce more than one value, such as structures, are not allowed. Character constants are denoted *'character'*. Numbers are viewed as integers unless a decimal point or exponent is used. In this case, they are treated as having the type double. Registers are viewed as integers. The *variable* may be an expression which indicates more than one variable, such as an array or structure name. If the address of a variable is given, it is regarded as the address of a variable of type *int*. C conventions are used in any type conversions necessary to perform the indicated assignment.

*x* Print the machine registers and the current machine-language instruction.



The commands for examining source files are:

*e procedure*  
*e file-name*  
*e directory/*  
*e directory file-name*

The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure name and file name are reported.

*/regular expression/*

Search forward from the current line for a line containing a string matching *regular expression*. Simple regular expression syntax may be used. The trailing */* may be omitted.

*? regular expression?*

Search backward from the current line for a line containing a string matching *regular expression*. Simple regular expression syntax may be used. The trailing *?* may be omitted.

*p* Print the current line.

*z* Print the current line followed by the next 9 lines. Set the current line to the last line printed.

*w* Window. Print the 10 lines around the current line.

*number*

Set the current line to the given line number. Print the new current line.

The commands for controlling the execution of the source program are:

*count r args*

*count R*

Run the program with the given arguments. The *r* command with no arguments reuses the previous arguments to the program while the *R* command runs the program with no arguments. An argument beginning with *<* or *>* causes redirection for the standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

*linenumber c count*

*linenumber C count*

Continue after a breakpoint or interrupt. If *count* is given, the program will stop when *count* breakpoints have been encountered. With the *C* command, the signal which caused the program to stop is reactivated; with the *c* command, it is ignored. If a line number is specified then a temporary breakpoint is placed at the line and execution is continued. The breakpoint is deleted when the command finishes. (It may not be possible to set breakpoints in some places, e.g., with shared libraries.)

**s** *count*

**S** *count*

Single step the program through *count* lines. If no count is given then the program is run for one line. **S** is equivalent to **s** except it steps through procedure calls.

**i**

**I** Single step by one machine-language instruction. With the **/** command, the signal which caused the program to stop is reactivated; with the **i** command, it is ignored.

**k** Kill the program being debugged.

*procedure(arg1,arg2,...)*

*procedure(arg1,arg2,...)/m*

Execute the named procedure with the given arguments. Arguments can be integer, character or string constants or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to *d*.

*linenumber b commands*

Set a breakpoint at the given line. If a procedure name without a line number is given (e.g., *proc:*), a breakpoint is placed at the first line in the procedure even if it was not compiled with the **-g** option. If no *linenumber* is given, a breakpoint is placed at the current line. (It may not be possible to set breakpoints in some places, e.g., with shared libraries.)

If no *commands* are given, execution stops just before the breakpoint and control is returned to *sdb*. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If **k** is used as a command to execute at a breakpoint, control returns to *sdb*, instead of continuing execution.

**B** Print a list of the currently active breakpoints.

*linenumber d*

Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d** then the breakpoint is deleted.

**D** Delete all breakpoints.

**I** Print the last executed line.



Miscellaneous commands:

*!command*

The command is interpreted by the command interpreter.

<*newline*>

If the previous command printed a source line, then advance the current line by one line and print the new current line. If the previous command displayed a memory location, then display the next memory location.

<EOF>

Scroll. Print the next 10 lines of instructions, source or data depending on which was printed last.

Read commands from *filename* until the end of file is reached, and then continue to accept commands from standard input. When *sdb* is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

" *string*

Print the given string. The C escape sequences of the form *\character* are recognised, where *character* is a nonnumeric character.

q Exit the debugger.

## FILES

a.out	default object file
core	default core file

## SEE ALSO

cc(1D).

## APPLICATION USAGE

By their very nature, symbolic debuggers are strongly system-dependent. Although *sdb* is optional in the Guide, strong efforts will be made to provide it, or similar functionality. Application developers should look at the documentation for their current system; common alternatives are *adb*(1), *dbx*(1) and *cdb*(1).

## CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

This utility is not optional in the SVID.

All references to F77 have been removed, and the words "and possibly programs in other languages" added to the first sentence of the **DESCRIPTION**.

The description of how to print variables has been corrected.

## NAME

sed — stream editor

## SYNOPSIS

sed [—n] [—e script] [—f sfile] [file...]

## DESCRIPTION

The command *sed* copies the named *files* (standard input default) to the standard output, edited according to a script of commands. The *—f* option causes the script to be taken from file *sfile*; these options accumulate. If there is just one *—e* option and no *—f* options, the flag *—e* may be omitted. The *—n* option suppresses the default output. A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation, *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a *D* command), applies in sequence all commands whose addresses select that pattern space, and at the end of the script copies the pattern space to the standard output (except under *—n*) and deletes the pattern space.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval.

An address is either a decimal number that counts input lines cumulatively across files, a *\$* that addresses the last line of input, or a context address, i.e., a */regular expression/* in the style of the *ed* command modified as follows:

In a context address, the construction *\?regular expression?*, where *?* is any character, is identical to */regular expression/*. Note that in the context address *\xabc\xdefx*, the second *x* stands for itself, so that the regular expression is *abcxdef*.

The escape sequence *\n* matches a newline embedded in the pattern space.

A period *.* matches any character except the terminal newline of the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Simple regular expression syntax is used throughout.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function *!* (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.



The argument *text* consists of one or more lines, all but the last of which end with \ to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an *s* command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line. The argument *rfile* or the argument *wfile* must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 10 distinct *wfile* arguments.

(1)a \

*text* Append. Place *text* on the output before reading the next input line.

(2)b *label*

Branch to the : command bearing the *label*. If *label* is empty, branch to the end of the script.

(2)c \

*text* Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.

(2)d Delete the pattern space. Start the next cycle.

(2)D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2)g Replace the contents of the pattern space by the contents of the hold space.

(2)G Append the contents of the hold space to the pattern space.

(2)h Replace the contents of the hold space by the contents of the pattern space.

(2)H Append the contents of the pattern space to the hold space.

(1)i \

*text* Insert. Place *text* on the standard output.

OF

(2)l List the pattern space on the standard output in an unambiguous form. Non-printing characters are spelled in two-digit ASCII and long lines are folded.

(2)n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2)N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2)p Print. Copy the pattern space to the standard output.

(2)P Copy the initial segment of the pattern space through the first newline to the standard output.

(1)q Quit. Branch to the end of the script. Do not start a new cycle.

(2)r *rfile*

Read the contents of *rfile*. Place them on the output before reading the next input line.

(2)s/*regular expression*/*replacement*/*flags*

Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character may be used instead of /. For a fuller description see the s command of ed(1). The value of *flags* is zero or more of:

n *n* = 1-512. Substitute for just the *n*th occurrence of the *regular expression*.

g Global. Substitute for all nonoverlapping instances of the *regular expression* rather than just the first one.

p Print the pattern space if a replacement was made.

w *wfile*

Write. Append the pattern space to *wfile* if a replacement was made.

(2)t *label*

Test. Branch to the : command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a t. If *label* is empty, branch to the end of the script.

(2)w *wfile*

Write. Append the pattern space to *wfile*.

(2)x Exchange the contents of the pattern and hold spaces.

(2)y/*string1*/*string2*/

Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal.

(2)! *function*

Don't. Apply the *function* (or group, if *function* is { }) only to lines not selected by the address(es).

(0): *label*

This command does nothing; it bears a *label* for b and t commands to branch to.

(1)= Place the current line number on the standard output as a line.

(2){ Execute the following commands through a matching } only when the pattern space is selected.



- (0) An empty command is ignored.
- (0) If a `#` appears as the first character on the first line of a script file, then that entire line is treated as a comment, with one exception. If the character after the `#` is an `n`, then the default output will be suppressed. The rest of the line after `n` is also ignored. A script file must contain at least one non-comment line.

## SEE ALSO

`awk(1)`, `ed(1)`, `grep(1)`.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

sh — shell, the command interpreter

## SYNOPSIS

sh [ flags ] [ arg ... ]

## DESCRIPTION

The command *sh* is a command interpreter that executes commands read from a terminal or a file. The command *sh -r* offers a restricted version of the command interpreter; it is used to set up login names and execution environments whose capabilities are more controlled. See *Invocation* below for the meaning of flags and other arguments to the shell.

## Definitions

A *blank* is a tab or a space.

A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore.

A *parameter* is a name, a digit, or any of the characters \*, @, #, ?, -, \$, and !.

## Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec(2)*). The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) 0200+status if it terminates abnormally. See *APPLICATION USAGE*, below, for a list of status values.

A *pipeline* is a sequence of one or more commands separated by |. The standard output of each command but the last is connected by a pipe, see *pipe(2)*, to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The *exit status* of a *pipeline* is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ; or &. Of these four symbols, ; and & have equal precedence, which is lower than that of && and ||. The symbols && and || also have equal precedence. The symbol ; causes sequential execution of the preceding pipeline; the symbol & causes asynchronous execution of the preceding pipeline (i.e., the shell does not wait for that pipeline to finish). The symbol && (||) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of newlines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a *simple-command* or one of the following. Unless otherwise stated, the *value* returned by a *command* is that of the last *simple-command* executed in the command.



**for name [ in word ... ] do list done**

Each time a *for* command is executed, *name* is set to the next *word* taken from the *in word* list. If *in word ...* is omitted, then the *for* command executes the *do* list once for each positional parameter that is set (see *Parameter Substitution* below). Execution ends when there are no more words in the list.

**case word in [ pattern [ | pattern ] ... ) list ;; ] ... esac**

A *case* command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see *File Name Generation*) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

**if list then list [ elif list then list ] ... [ else list ] fi**

The *list* following *if* is executed and, if it returns a zero exit status, the *list* following the first *then* is executed. Otherwise, the *list* following *elif* is executed and, if its value is zero, the *list* following the next *then* is executed. Failing that, the *else* list is executed. If no *else* list or *then* list is executed, the *if* command returns a zero exit status.

**while list do list done**

A *while* command repeatedly executes the *while* list and, if the exit status of the last command in the list is zero, executes the *do* list; otherwise the loop terminates. If no commands in the *do* list are executed, then the *while* command returns a zero exit status; *until* may be used in place of *while* to negate the loop termination test.

**( list )**

Execute *list* in a sub-shell.

**{ list ; }**

*list* is simply executed. (The semi-colon may be replaced by a newline.)

**name() { list; }**

Define a function which is referenced by *name*. The body of the function is the *list* of commands between { and }. (The semi-colon may be replaced by a newline.) Execution of functions is described below (see *Execution*).

The following words are only recognised as the first word of a command and when not quoted:

```
if then else elif fi case
esac for while until do done { }
```

**Comments**

A word beginning with # causes that word and all the following characters up to a newline to be ignored.

**Command Substitution**

The standard output from a command enclosed in a pair of grave accents (`) may be used as part or all of a word; trailing newlines are removed.

### Parameter Substitution

The character `$` is used to introduce substitutable parameters. There are two types of parameters; positional and keyword. If the parameter name is a single digit (0 - 9), it is a positional parameter; otherwise, the name must be a legal *name* as defined above, and gives a keyword parameter. Positional parameters may be assigned values by *set*. Keyword parameters (also known as variables) may be assigned values by writing:

```
name=value [ name=value ] ...
```

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

#### `${parameter}`

The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is `*` or `@`, all the positional parameters, starting with `$1`, are substituted (separated by spaces). Parameter `$0` is set from argument zero when the shell is invoked.

#### `${parameter:-word}`

If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*.

#### `${parameter:=word}`

If *parameter* is not set or is null, set it to *word*; the value of the parameter is substituted. Positional parameters may not be assigned to in this way.

#### `${parameter:?word}`

If *parameter* is set and is non-null, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, the message *parameter null or not set* is printed.

#### `${parameter:+word}`

If *parameter* is set and is non-null, substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string so that, in the following example, *pwd* is executed only if *d* is not set or is null:

```
echo ${d:-`pwd`}
```

If the colon (`:`) is omitted from the above expressions, the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

- `#` The number of positional parameters in decimal.
- `-` Flags supplied to the shell on invocation or by the *set* command.
- `?` The decimal value returned by the last synchronously executed command.
- `$` The process number of this shell.
- `!` The process number of the last background command invoked.

The following parameters are used by the shell:

`HOME` The default argument (home directory) for the *cd* command.



**PATH** The search path for commands (see *Execution* below). The user may not change **PATH** if executing in restricted mode.

**CDPATH**

The search path for the *cd* command. The syntax and usage is similar to that of **PATH**.

**MAIL** If this parameter is set to the name of a mail file, then the shell informs the user of the arrival of mail in the specified file. The user is informed only if **MAIL** is set and **MAILPATH** is not set.

**MAILCHECK**

This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each primary prompt.

**MAILPATH**

A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes.

**PS1** Primary prompt string, by default \$.

**PS2** Secondary prompt string, by default >.

**IFS** Internal field separators, normally <space>, <tab>, and <newline>.

**SHACCT**

If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed.

**SHELL**

When the shell is invoked, it scans the environment (see *Environment* below) for this name. If it is found and there is an *r* in the file name part of its value, the shell becomes a restricted shell.

The shell gives default values to **PATH**, **PS1**, **PS2**, **MAILCHECK** and **IFS**.

**Blank Interpretation**

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (those found in **IFS**) and split into distinct arguments where such characters are found. Explicit null arguments ( " " or `` ) are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.

**File Name Generation**

Following substitution, each command *word* is scanned for the shell metanotation characters \*, ? and [. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character . at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

### Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

`; & ( ) | ^ < > <newline> <space> <tab>`

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a `\`. The pair `\<newline>` is ignored. All characters enclosed between a pair of single quote marks (`' '`), except a single quote, are quoted. Inside double quote marks (`" "`), parameter and command substitution occurs and `\` quotes the characters `\`, `'`, `"`, and `$`. `"$*"` is equivalent to `"$1 $2 ..."`, whereas `"$@"` is equivalent to `"$1" "$2" ...`.

### Prompting

When used interactively, the shell prompts with the value of `PS1` before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt (i.e., the value of `PS2`) is issued.

### Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a *simple-command* or may precede or follow a command and are not passed on to the invoked command; substitution occurs before *word* or *digit* is used:

`<word`

Use file *word* as standard input (file descriptor 0).

`>word`

Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise it is truncated to zero length.

`>>word`

Use file *word* as standard output. If the file exists output is appended to it; otherwise the file is created.

`<<[-] word`

The shell input is read up to a line that is the same as *word* or to an end-of-file. The resulting document becomes the standard input. If any character of *word* is quoted, no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, (unescaped) `\<newline>` is ignored, and `\` must be used to quote the characters `\`, `$`, `'`, and the first character of *word*. If `-` is appended to `<<`, all leading tabs are stripped from *word* and from the document.

`<&digit`

Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using `>&digit`.

`<&-` The standard input is closed. Similarly for the standard output using `>&-`.



If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

```
... 1>xxx 2>&1
```

first associates file descriptor 1 with the file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e., *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

If a command is followed by *&* the default standard input for the command is the empty file */dev/null*. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Redirection of output is not allowed in the restricted shell.

#### Environment

The *environment* is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the *export* command is used to bind the shell's parameter to the environment (see also *set —a*). A parameter may be removed from the environment with the *unset* command. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by *unset*, plus any modifications or additions, all of which must be noted in *export* commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. Thus:

```
TERM=123 cmd  
(export TERM; TERM=123; cmd)
```

(where *cmd* uses the value of the environment variable *TERM*) are equivalent as far as the execution of *cmd* is concerned.

If the *—k* flag is set, all keyword arguments are placed in the environment, even if they occur after the command name. The following first prints *a=b c* and *c*:

```
echo a=b c  
set —k  
echo a=b c
```

### Signals

The **SIGINT** and **SIGQUIT** signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent (but see also the *trap* command below).

### Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the **Special Commands** listed below, it is executed in the shell process. If the command name does not match a **Special Command**, but matches the name of a defined function, the function is executed in the shell process (note how this differs from the execution of shell procedures). The positional parameters *\$1*, *\$2*, ... are set to the arguments of the function. If the command name matches neither a **Special Command** nor the name of a defined function, a new process is created and an attempt is made to execute the command via an *exec* routine, see *exec(2)*.

The variable *PATH* defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between two colon delimiters anywhere else in the path list. If the command name contains a / the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory from left to right in the path is searched in turn for an executable file of the name of the command. If the file has execute permission but is not an executable file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. A parenthesised command is also executed in a sub-shell.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *exec* calls later). If the command was found in a relative directory, its location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the *PATH* variable is changed or the *hash -r* command is executed (see below).

### Special Commands

**:** Null effect; the command does nothing. Arguments are parsed as usual. A zero exit code is returned.

**. file**

Read and execute commands from *file* and return. The search path specified by *PATH* is used to find the directory containing *file*.

**break [ n ]**

Exit from the enclosing *for* or *while* loop, if any. If *n* is specified break *n* levels.

**continue [ n ]**

Resume the next iteration of the enclosing *for* or *while* loop. If *n* is specified resume at the *n*th enclosing loop.



**cd** [ *arg* ]

Change the current directory to *arg*. The variable HOME is the default *arg*. The variable CDPATH defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is *null* (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / the search path is not used. Otherwise, each directory in the path is searched for *arg* and the name of the directory selected is printed. The *cd* command may not be executed in restricted mode.

**echo** [ *arg* ... ]

Echo arguments. See *echo*(1) for usage and description.

**eval** [ *arg* ... ]

The arguments are read as input to the shell and the resulting command(s) executed.

**exec** [ *arg* ... ]

The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

**exit** [ *n* ]

Causes a shell to exit with the exit status specified by *n*. If *n* is omitted the exit status is that of the last command executed (an end-of-file will also cause the shell to exit).

**export** [ *name* ... ]

The given *names* are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, a list of all names that are exported in this shell is printed. Function names may not be exported.

**hash** [ *—r* | *name* ... ]

For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The *—r* option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented.

**pwd** Print the current working directory. See *pwd*(1) for usage and description.

**read** [ *name* ... ]

One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. Only the characters in the variable IFS are recognised as delimiters. The return code is 0 unless an end-of-file is encountered.

**readonly** [ *name* ... ]

The given *names* are marked *readonly* and the values of these *names* may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed.

**return** [ *n* ]

Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

**set** [ — —aefhkntuvx [ *arg* ... ] ]

- a Mark variables which are modified or created for export.
- e Exit immediately if a command exits with a non-zero exit status.
- f Disable file name generation.
- h Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).
- k All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- n Read commands but do not execute them.
- t Exit after reading and executing one command.
- u Treat unset variables as an error when substituting.
- v Print shell input lines as they are read.
- x Print commands and their arguments as they are executed.
- — Do not change any of the flags; useful in setting \$1 to —.

Using + rather than — causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in \$-. The remaining arguments are positional parameters and are assigned, in order, to \$1, \$2, .... If no arguments are given the values of all names are printed.

**shift** [ *n* ]

The positional parameters from \$*n*+1 are renamed \$1.... If *n* is not given, it is assumed to be 1.

**test** Evaluate conditional expressions. See *test*(1) for usage and description.

**times**

Print the accumulated user and system times for processes run from the shell.



**trap** [ *arg* ] [ *n* ] ...

The command *arg* is to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If *arg* is absent all trap(s) *n* are reset to their original values. If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *arg* is executed on exit from the shell. The *trap* command with no arguments prints a list of commands associated with each signal number.

**type** [ *name* ... ]

For each *name*, indicate how it would be interpreted if used as a command name.

**ulimit** [ *-f n* ]

If the *-f n* option is used, then a size limit of *n* 512-byte units is imposed on files written by the shell and its child processes (files of any size may be read). Only the superuser may increase the limit. If *n* is omitted, the current limit is printed. If no option is given, *-f* is assumed.

**umask** [ *ooo* ]

The user file-creation mask is set to the octal value *ooo*, see *umask*(2). If *ooo* is omitted, the current value of the mask is printed.

**unset** [ *name* ... ]

For each *name*, remove the corresponding variable or function. The variables PATH, PS1, PS2, MAILCHECK and IFS cannot be unset.

**wait** [ *n* ]

Wait for the specified process and report its termination status. If *n* is not given all currently active child processes are waited for and the return code is zero.

#### Invocation

If the shell is invoked through an *exec* routine, see *exec*(2), and the first character of argument zero is *-*, commands are initially read from */etc/profile* and from *\$HOME/.profile*, if such files exist. Thereafter, commands are read as described below. The flags below are interpreted by the shell on invocation only; note that unless the *-c* or *-s* is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

*-c string*

If the *-c* flag is present commands are read from *string*.

*-s*

If the *-s* flag is present or if no arguments remain commands are read from the standard input. Any remaining arguments specify the positional parameters. Shell output (except for **Special Commands**) is written to file descriptor 2.

- i If the —i flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case SIGTERM is ignored (so that *kill 0* does not kill an interactive shell) and SIGINT is caught and ignored (so that *wait* is interruptible). In all cases, SIGQUIT is ignored by the shell.
- r If the —r flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the *set* command above.

#### Restricted Shell

A restricted shell is used to set up login names and execution environments whose capabilities are more controlled. In a restricted shell, the following are disallowed:

- changing directory, see *cd*(1);
- setting the value of PATH;
- specifying command names containing /, and
- redirecting output (> and >>).

The restrictions above are enforced after *.profile* is interpreted.

When a command to be executed is found to be a shell procedure, an unrestricted shell is invoked to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the *.profile* has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably not the login directory).

#### FILES

/etc/profile	system wide startup file
\$HOME/.profile	per user startup file
/dev/null	null device

#### EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the *exit* command above).

#### SEE ALSO

*cd*(1), *echo*(1), *pwd*(1), *test*(1), *umask*(1), *dup*(2), *exec*(2), *exit*(2), *fork*(2), *pipe*(2), *signal*(2), *system*(2), *ulimit*(2), *umask*(2), *wait*(2).

#### APPLICATION USAGE

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. The *hash* command should be used to correct this situation.



If the current directory or the one above it is moved, *pwd* may not give the correct response. The *cd* command with a full path name should be used to correct this situation.

The output from the built-in commands should not be relied upon for use by other programs, as its format is not rigorously specified.

The following numeric values for the signals can be used in the *trap* command:

1	SIGHUP
2	SIGINT
3	SIGQUIT
9	SIGKILL
15	SIGTERM

Refer to *signal(2)* for the definitions of these signals.

If *sh* is invoked by the name *rsh*, typically via a link to *sh*, it behaves in the restricted way described under the *—r* option. However, the name *rsh* is used for a command in common use in networking extensions, which has different behaviour. For this reason, the description of *rsh* has been removed.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The sentence "Arguments are parsed as usual." has been added to the description of the *:* command.

The words "and the name of the directory selected is printed" have been added in the description of the *cd* command.

The sentence "Only the superuser may increase the limit." has been added to the description of the *ulimit* command.

The description of the *rsh* command has been removed from the description.

## NAME

shl — shell layer manager (OPTIONAL)

## SYNOPSIS

**shl**

OP UN

## DESCRIPTION

The command *shl* allows a user to interact with more than one shell from a single terminal. The user controls these shells, known as *layers*, using the commands described below.

The *current layer* is the layer which can receive input from the keyboard. Other layers attempting to read from the keyboard are blocked. Output from multiple layers is multiplexed onto the terminal. To have the output of a layer blocked when it is not current, the *stty* option *loblk* may be set within the layer.

The *stty* character *switch* (set to control-Z if NUL) is used to switch control to *shl* from a layer. The command *shl* has its own prompt, >>>, to help distinguish it from a layer.

A *layer* is a shell which has been bound to a virtual tty device (*/dev/sxt/\**). The virtual device can be manipulated like a real tty device using *stty* and *ioctl()*. See *stty(1)* and *ioctl(2)* respectively. Each layer has its own process group ID.

## Definitions

A *name* is a sequence of characters delimited by a blank, tab or newline. Only the first eight characters are significant. The names (1) through (7) cannot be used when creating a layer. They are used by *shl* when no name is supplied. They may be abbreviated to just the digit.

## Commands

The following commands may be issued from the *shl* prompt level. Any unique prefix is accepted.

**create** [ *name* ]

Create a layer called *name* and make it the current layer. If no argument is given, a layer will be created with a name of the form (#) where # is the last digit of the virtual device bound to the layer. The shell prompt variable PS1 is set to the name of the layer followed by a space. A maximum of seven layers can be created.

**block** *name* [ *name* ... ]

For each *name*, block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option *loblk* within the layer.

**delete** *name* [ *name* ... ]

For each *name*, delete the corresponding layer. All processes in the process group of the layer are sent the SIGHUP signal.

**help** (or ?)

Print the syntax of the *shl* commands.



**layers** [ **—l** ] [ *name* ... ]

For each *name*, list the layer name and its process group. The **—l** option produces a long listing. If no arguments are given, information is presented for all existing layers.

**resume** [ *name* ]

Make the layer referenced by *name* the current layer. If no argument is given, the last existing current layer will be resumed.

**toggle**

Resume the layer that was current before the last current layer.

**unblock** *name* [ *name* ... ]

For each *name*, do not block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option **—loblk** within the layer.

**quit** Exit *shl*. All layers are sent the SIGHUP signal.

*name* Make the layer referenced by *name* the current layer.

#### FILES

/dev/sxt/\*      Virtual tty devices

#### SEE ALSO

sh(1), stty(1), ioctl(2), signal(2), termio(7).

#### APPLICATION USAGE

Typical implementations of this utility require a communications line configured to use the *termio*(7) interface. On systems where none of these lines are available, this utility may not be present.

This utility also depends on the **optional** virtual tty devices and applications should be aware that these may not be present where these are not supported.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

This utility is not optional in the SVID.

## NAME

size — print section sizes of object files

## SYNOPSIS

size [—o] [—x] [—V] file...

## DESCRIPTION

The *size* command produces section size information for each section in the loaded object files. The sizes of the loaded sections are printed along with the sum of these sizes. If an archive file is input to the *size* command, the information for all archive members is displayed.

UN

Numbers will be printed in decimal unless either the —o or the —x option is used, in which case they will be printed in octal or in hexadecimal, respectively.

MV UN

The —V flag will supply the version information on the *size* command.

## SEE ALSO

cc(1D), ld(1D).

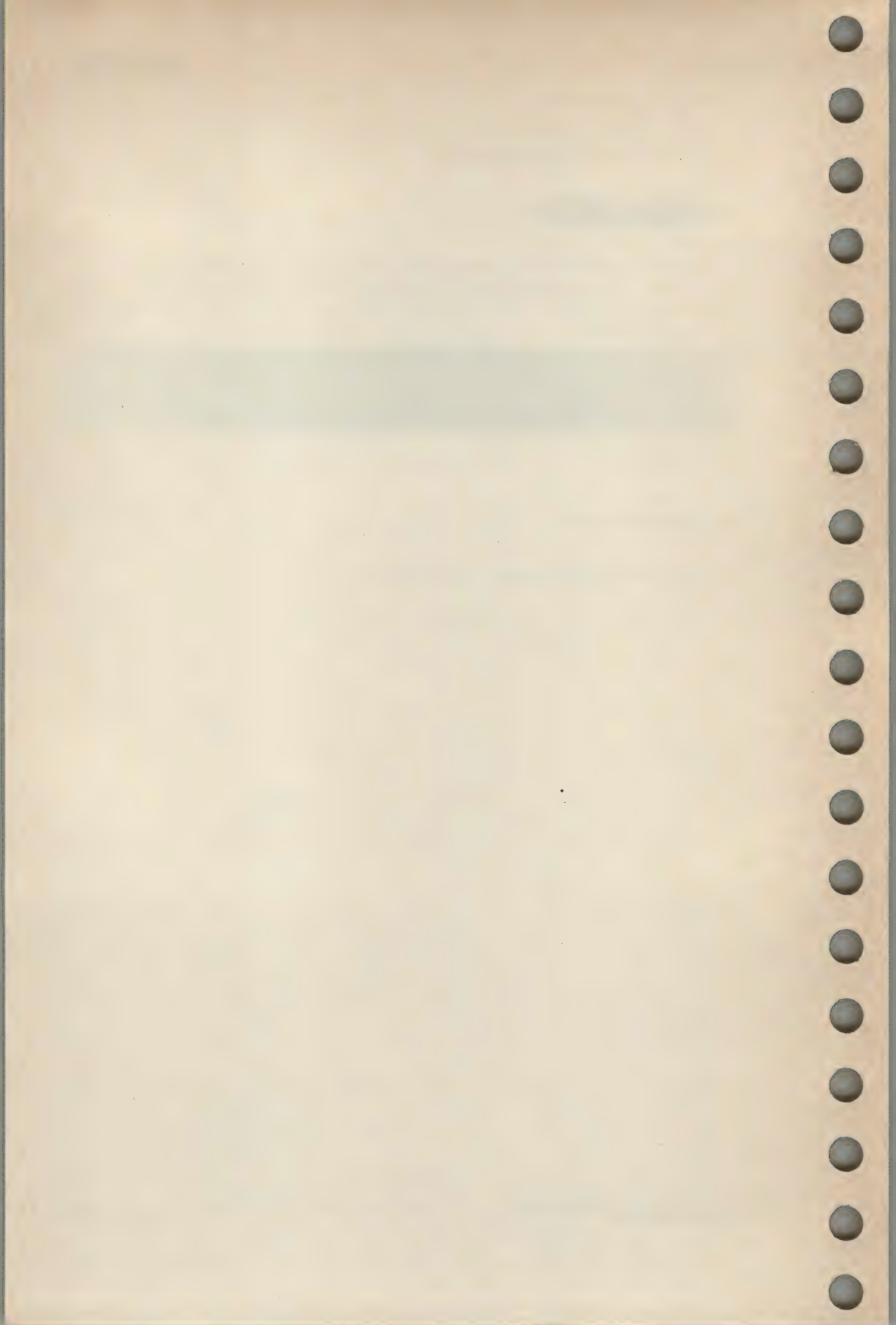
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

sleep — suspend execution for an interval

## SYNOPSIS

sleep time

## DESCRIPTION

The command *sleep* suspends execution for *time* seconds. It is used to execute a command after a certain amount of time, as in:

```
(sleep 105; command) &
```

or to execute a command every so often, as in:

```
while true
do
    command
    sleep 37
done
```

## SEE ALSO

alarm(2), sleep(2).

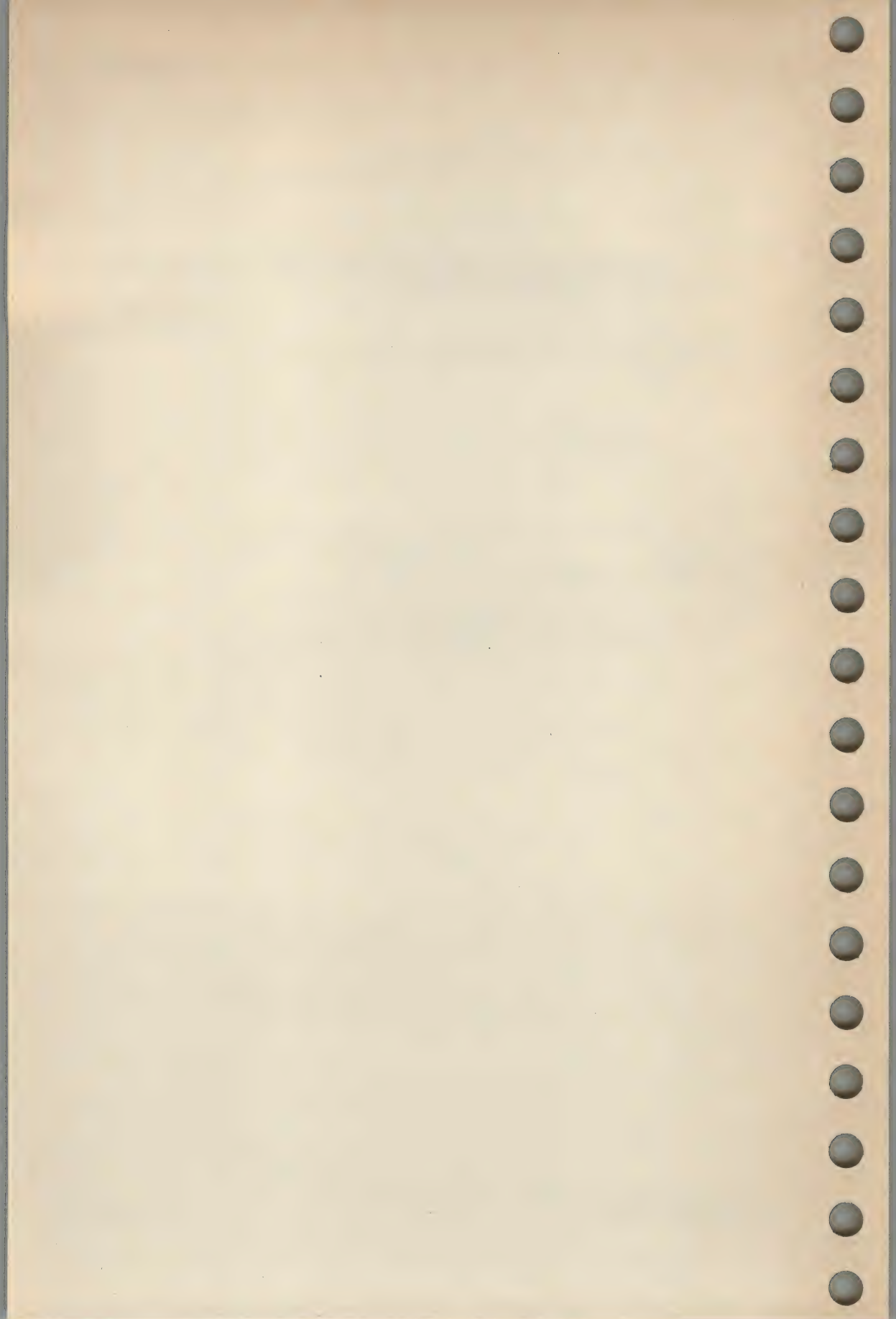
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

sort — sort and/or merge files

## SYNOPSIS

```
sort [—cmu] [—ooutput] [—ykmem] [—zrecsz] [—dfnr] [—btx]
    [+pos1 [—pos2]] [file...]
```

## DESCRIPTION

The command *sort* sorts lines of all the named files together and writes the result on the standard output. The standard input is read if *—* is used as a file name or no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

The following options alter the default behaviour:

- c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.
- m Merge only, the input files are already sorted.
- u Unique: suppress all but one in each set of lines having equal keys.
- ooutput

The argument given is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. There may be optional blanks between *—o* and *output*.

PI

## —y[kmem]

The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is a waste. If this option is omitted, *sort* begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, *kmem*, *sort* will start using that number of kilobytes of memory, unless the administrative minimum or maximum is violated, in which case the corresponding extremum will be used. Thus, *—y0* is guaranteed to start with minimum memory. By convention, *—y* (with no argument) starts with maximum memory.

## —zrecsz

The size of the longest line read in the sort phase is recorded so that buffers of the correct size can be allocated during the merge phase. If the sort phase is omitted via the *—c* or *—m* options, a system dependent default size will be used. Lines longer than the buffer size will cause *sort* to terminate abnormally. Supplying the actual number of bytes in the longest line to be merged (or some larger value) will prevent abnormal termination.

The following options override the default ordering rules.

- d Only US ASCII letters, digits and blanks (spaces and tabs) are significant in comparisons.



- f Ignore case of US ASCII letters.
- i Ignore characters outside the range 040-0176 in non-numeric comparisons.
- n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The —n option implies the —b option (see below). Note that the —b option is only effective when restricted sort key specifications are in effect.
- r Reverse the sense of comparisons.

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

The notation *+pos1 —pos2* restricts a sort key to one beginning at *pos1* and ending at *pos2*. The characters at positions *pos1* and *pos2* are included in the sort key (provided that *pos2* does not precede *pos1*). A missing *—pos2* means the end of the line.

Specifying *pos1* and *pos2* involves the notion of a field, a minimal sequence of characters followed by a field separator or a newline. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field. The treatment of field separators can be altered using the options:

- tx Use *x* as the field separator character; *x* is not considered to be part of a field (although it may be included in a sort key). Each occurrence of *x* is significant (e.g., *xx* delimits an empty field).
- b Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the —b option is specified before the first *+pos1* argument, it will be applied to all *+pos1* arguments. Otherwise, the *b* flag may be attached independently to each *+pos1* or *—pos2* argument (see below).

The arguments *pos1* and *pos2* each have the form *m.n* optionally followed by one or more of the flags *bdfnr*. A starting position specified by *+m.n* is interpreted to mean the *n*+1th character in the *m*+1th field. A missing *.n* means *.0*, indicating the first character of the *m*+1th field. If the *b* flag is in effect *n* is counted from the first non-blank in the *m*+1th field; *+m.0b* refers to the first non-blank character in the *m*+1th field.

A last position specified by *—m.n* is interpreted to mean the *n*th character (including separators) after the last character of the *m*th field. A missing *.n* means *.0*, indicating the last character of the *m*th field. If the *b* flag is in effect *n* is counted from the last leading blank in the *m*+1th field; *—m.1b* refers to the first non-blank in the *m*+1th field.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

## EXAMPLES

1. Sort the contents of *infile* with the second field as the sort key:  
`sort +1 —2 infile`
2. Sort, in reverse order, the contents of *infile1* and *infile2*, placing the output in *outfile* and using the first character of the second field as the sort key:  
`sort —r —o outfile +1.0 —1.2 infile1 infile2`
3. Sort, in reverse order, the contents of *infile1* and *infile2* using the first non-blank character of the second field as the sort key:  
`sort —r +1.0b —1.1b infile1 infile2`
4. Print the password file sorted by the numeric user ID (the third colon-separated field):  
`sort —t: +2n —3 /etc/passwd`
5. Print the lines of the already sorted file *infile*, suppressing all but the first occurrence of lines having the same third field (the options `—um` with just one input file make the choice of a unique representative from a set of equal lines predictable):  
`sort —um +2 —3 infile`

## EXIT STATUS

*Sort* comments and exits with non-zero status for various trouble conditions (e.g., when input lines are too long), and for disorder discovered under the `—c` option.

When the last line of an input file is missing a newline character, *sort* appends one, prints a warning message, and continues.

## SEE ALSO

`comm(1)`, `join(1)`, `uniq(1)`.

## APPLICATION USAGE

The user is warned that the semantics of the `+pos1 —pos2` are different in earlier versions of *sort(1)*.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

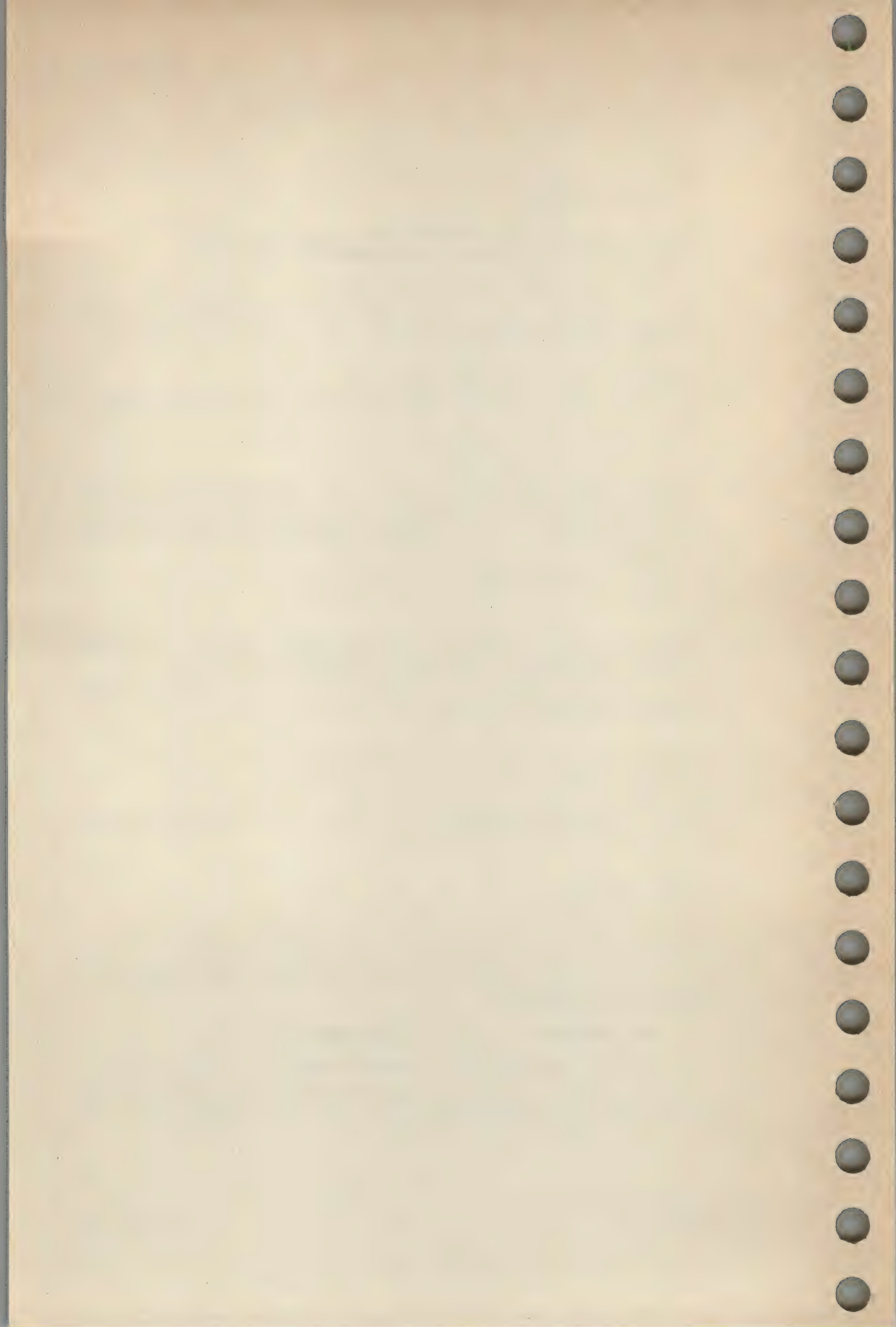
The word "Dictionary" has been removed from the description of the `—d` flag as it is not valid in many parts of the world.

The word "ASCII" has been removed from the description of the `—i` option.

The words "US ASCII" have been added to the descriptions of the `—f` and `—d` flags.

In the description of the `—f` option, the words "Fold lower case US ASCII letters into upper case" have been changed to read "Ignore case of US ASCII letters".





## NAME

spell — find spelling errors

## SYNOPSIS

```
spell [ -v ] [ -b ] [ -x ] [ +local_file ] [ file... ]
```

OF LA

## DESCRIPTION

The command *spell* collects words from the named *files* and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no *files* are named, words are collected from the standard input.

Under the *-v* option, all words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.

Under the *-b* option, British spelling is checked. Besides preferring *centre*, *colour*, *programme*, *speciality*, *travelled*, etc., this option insists upon *-ise* in words like *standardise*.

Under the *-x* option, every plausible stem is printed with = for each word.

Under the *+local\_file* option, words found in *local\_file* are removed from *spell*'s output. The argument *local\_file* is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to *spell*'s own spelling list) for each job.

## FUTURE DIRECTIONS

In order to conform to the command syntax standard, the *+local\_file* option will be changed to the form *-flocal\_file*. The old form will continue to be accepted for some time.

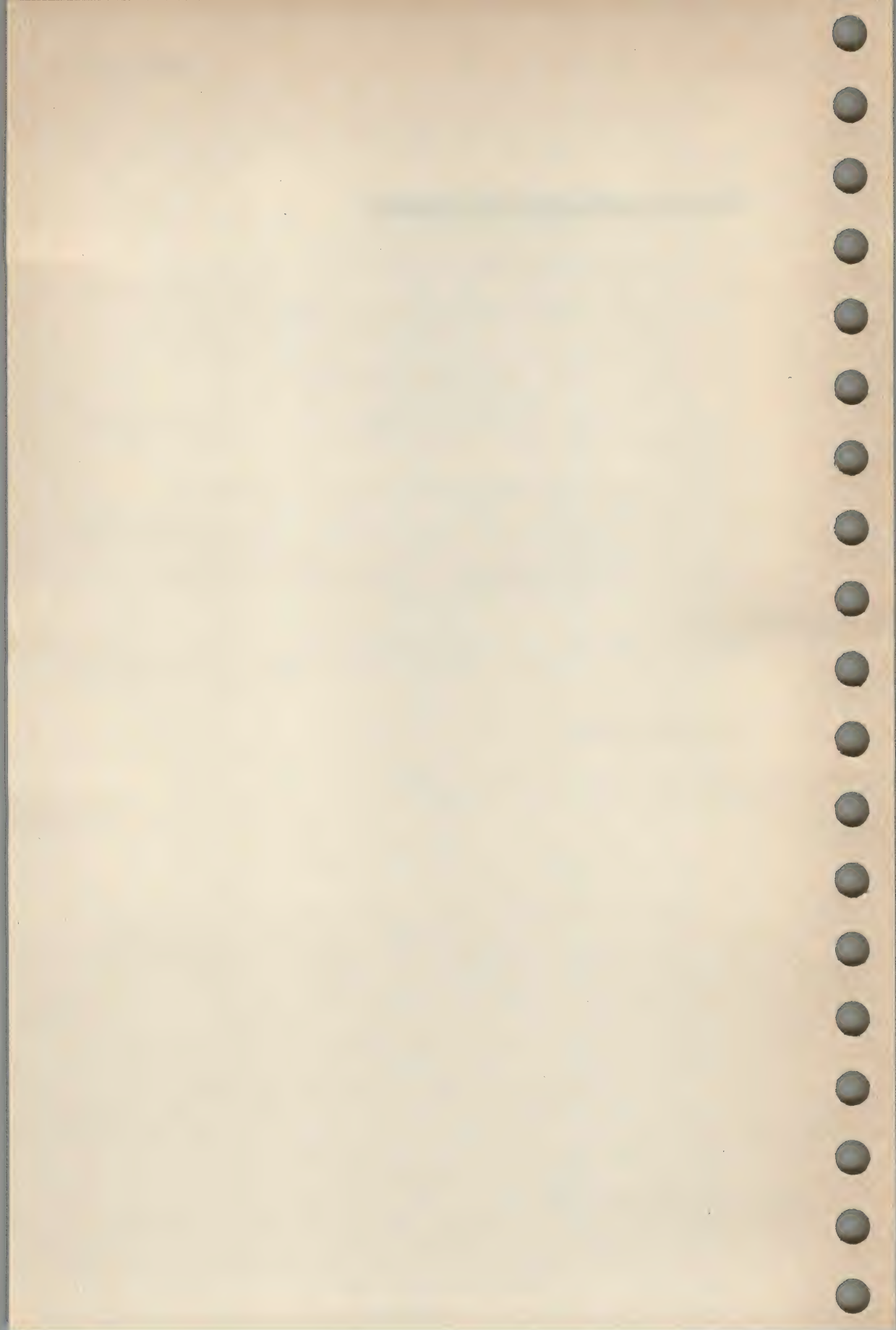
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





**NAME**

`split` — split a file into pieces

**SYNOPSIS**

`split [ —n ] [ file [ name ] ]`

**DESCRIPTION**

The command *split* reads *file* and writes it in *n-line* pieces (default 1000 lines) onto a set of output files. The name of the first output file is *name* with *aa* appended, and so on lexicographically, up to *zz* (a maximum of 676 files). The argument *name* cannot be longer than {NAME\_MAX}-2 characters. If no output name is given, *x* is default.

If no input file is given, or if `—` is given in its stead, then the standard input is used.

**SEE ALSO**

`csplit(1)`.

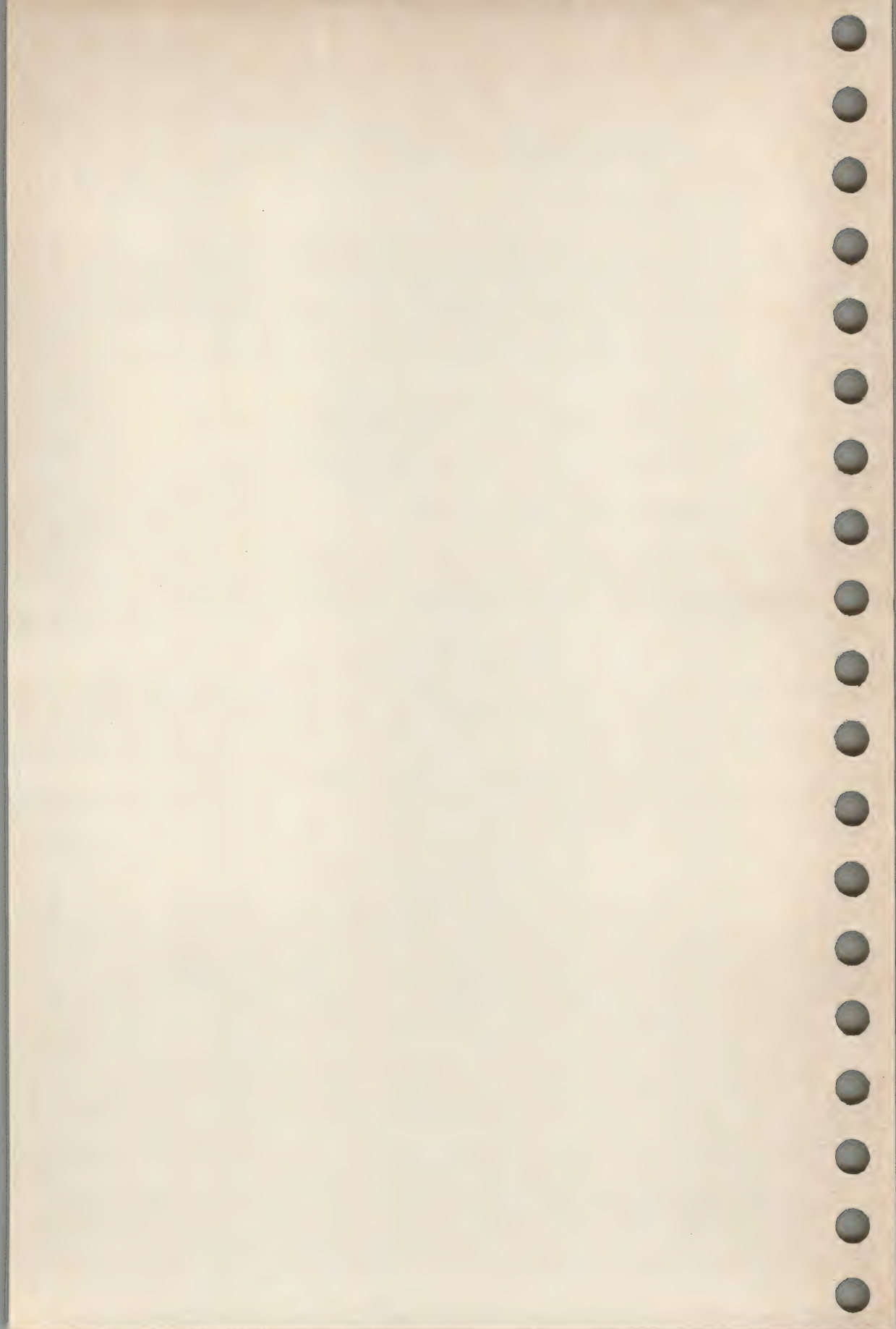
**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID.





## NAME

strip — strip symbolic information from an object file

## SYNOPSIS

strip [-x] [-r] [-V] file...

## DESCRIPTION

The *strip* command strips the symbolic information from object files or archives of object files.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

- MV UN -x Do not strip static or external symbol information.
- MV UN -r Do not strip static or external symbol information, or relocation information.
- MV UN -V Print the version of the *strip* command, on the standard error output.

If there is any relocation information in the object file and any symbol table information is to be stripped, *strip* will report an error and terminate without stripping *file* unless the *-r* flag is used.

## SEE ALSO

ar(1), cc(1D), ld(1D).

## APPLICATION USAGE

The purpose of this command is to reduce the file storage overhead taken by the object file.

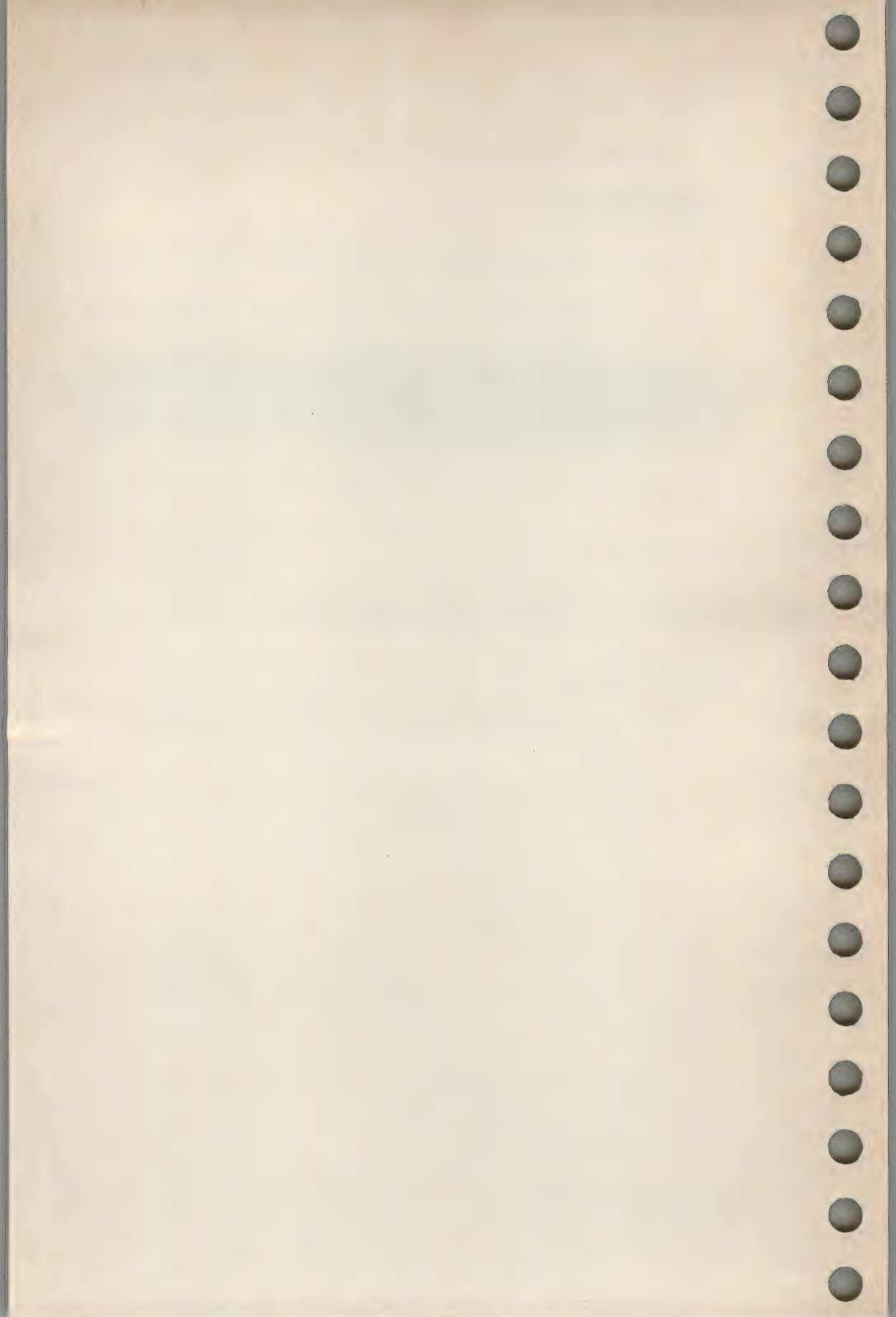
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

stty — set the options for a terminal

## SYNOPSIS

stty [**—a**] [**—g**] [**options**]

UN

## DESCRIPTION

The command *stty* sets certain terminal I/O options for the device that is its standard input; without arguments, it reports the settings of certain options; with the **—a** option, it reports all of the option settings; with the **—g** option, it reports current settings in a form that can be used as an argument to another *stty* command. Detailed information about the modes listed in the first five groups below may be found in *ioctl(2)*. Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sanity checking is performed. The options are selected from the following:

## Control Modes

**parenb** (**—parenb**)

enable (disable) parity generation and detection.

**parodd** (**—parodd**)

select odd (even) parity.

**cs5 cs6 cs7 cs8**

select character size.

**0** hang up “phone” line immediately. This applies to all terminal lines — not just modem lines. A SIGHUP signal is sent to all processes attached to the terminal.

*number*

set terminal baud rate to the *number* given, if possible. (All speeds are not supported by all hardware interfaces.)

**hupcl** (**—hupcl**)

hang up (do not hang up) modem connection on last close.

**hup** (**—hup**)

same as *hupcl* (**—hupcl**).

**cstopb** (**—cstopb**)

use two (one) stop bits per character.

**cread** (**—cread**)

enable (disable) the receiver.

**clocal** (**—clocal**)

assume a line without (with) modem control.

**loblk** (**—loblk**)

block (do not block) output from a non-current layer.

## Input Modes

**ignbrk** (**—ignbrk**)

ignore (do not ignore) break on input.



- brkint** (**—brkint**)  
signal (do not signal) SIGINT on break.
- ignpar** (**—ignpar**)  
ignore (do not ignore) parity errors.
- parmrk** (**—parmrk**)  
mark (do not mark) parity errors.
- inpck** (**—inpck**)  
enable (disable) input parity checking.
- istrip** (**—istrip**)  
strip (do not strip) input characters to seven bits.
- inlcr** (**—inlcr**)  
map (do not map) <newline> to <carriage return> on input.
- igncr** (**—igncr**)  
ignore (do not ignore) <carriage return> on input.
- icrnl** (**—icrnl**)  
map (do not map) <carriage return> to <newline> on input.
- iuclic** (**—iuclic**)  
map (do not map) upper-case alphabets to lower case on input.
- ixon** (**—ixon**)  
enable (disable) START/STOP output control. Output is stopped by sending an ASCII DC3 and started by sending an ASCII DC1.
- ixany** (**—ixany**)  
allow any character (only DC1) to restart output.
- ixoff** (**—ixoff**)  
request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

#### Output Modes

- opost** (**—opost**)  
post-process output (do not post-process output; ignore all other output modes).
- olcuc** (**—olcuc**)  
map (do not map) lower-case alphabets to upper case on output.
- onlcr** (**—onlcr**)  
map (do not map) <newline> to <carriage return-newline> on output.
- ocrnl** (**—ocrnl**)  
map (do not map) <carriage return> to <newline> on output.
- onocr** (**—onocr**)  
do not (do) output <carriage return>s at column zero.

**onlret (—onlret)**

on the terminal <newline> performs (does not perform) the function.

**ofill (—ofill)**

use fill characters (use timing) for delays.

**ofdel (—ofdel)**

fill characters are DELs (NULs).

**cr0 cr1 cr2 cr3**

select style of delay for carriage returns.

**nl0 nl1**

select style of delay for line-feeds.

**tab0 tab1 tab2 tab3**

select style of delay for horizontal tabs.

**bs0 bs1**

select style of delay for backspaces.

**ff0 ff1**

select style of delay for form-feeds.

**vt0 vt1**

select style of delay for vertical tabs.

**Local Modes****isig (—isig)**

enable (disable) the checking of characters against the special control characters INTR, QUIT, and SWITCH.

**icanon (—icanon)**

enable (disable) canonical input (ERASE and KILL processing).

**xcase (—xcase)**

canonical (unprocessed) upper/lower-case presentation.

**echo (—echo)**

echo back (do not echo back) every character typed.

**echoe (—echoe)**

echo (do not echo) ERASE character as a backspace-space-backspace string.

**Note:** this mode will erase the ERASEed character on many CRT terminals; however, it does not keep track of column position and, as a result, may be confusing on escaped characters, tabs, and backspaces.

**echok (—echok)**

echo (do not echo) <newline> after KILL character.

**echonl (—echonl)**

echo (do not echo) <newline>.

**noflsh (—noflsh)**

disable (enable) flush after INTR, QUIT, or SWITCH.



## Control Assignments

*control-character c*

set *control-character* to *c*, where *control-character* is ERASE, KILL, INTR, QUIT, SWTCH, EOF, EOL, MIN, or TIME (MIN and TIME are used with *—icanon*). If *c* is preceded by a caret (^), then the value used is the corresponding CTRL character (e.g., “^D” is a CTRL-D); “^?” is interpreted as DEL and “^—” is interpreted as undefined.

*line i*

set line discipline to *i* ( $0 < i < 127$ ).

## Combination Modes

*evenp or parity*

enable *parenb* and *cs7*.

*oddp*

enable *parenb*, *cs7*, and *parodd*.

*—parity , —evenp or —oddp*

disable *parenb*, and set *cs8*.

*raw (—raw or cooked)*

enable (disable) raw input and output (equivalent to *cs8*, with no ERASE, KILL, INTR, QUIT, SWTCH, EOT, or output post processing, and no parity).

*nl (—nl)*

unset (set) *icrnl*, *onlcr*. In addition *—nl* unsets *inlcr*, *igncr*, *ocrnl*, and *onlret*.

*lcase (—lcase)*

set (unset) *xcase*, *iuc lc*, and *olcuc*.

*LCASE (—LCASE)*

same as *lcase (—lcase)*.

*tabs (—tabs or tab8)*

preserve (expand to spaces) tabs when printing.

*ek*

reset ERASE and KILL characters back to the system defaults.

*sane*

resets all modes to some reasonable values.

## SEE ALSO

*ioctl(2)*, *termio(7)*.

## APPLICATION USAGE

Typical implementations of this utility require a communications line configured to use the *termio(7)* interface. On systems where none of these lines are available, this utility may not be present.

CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

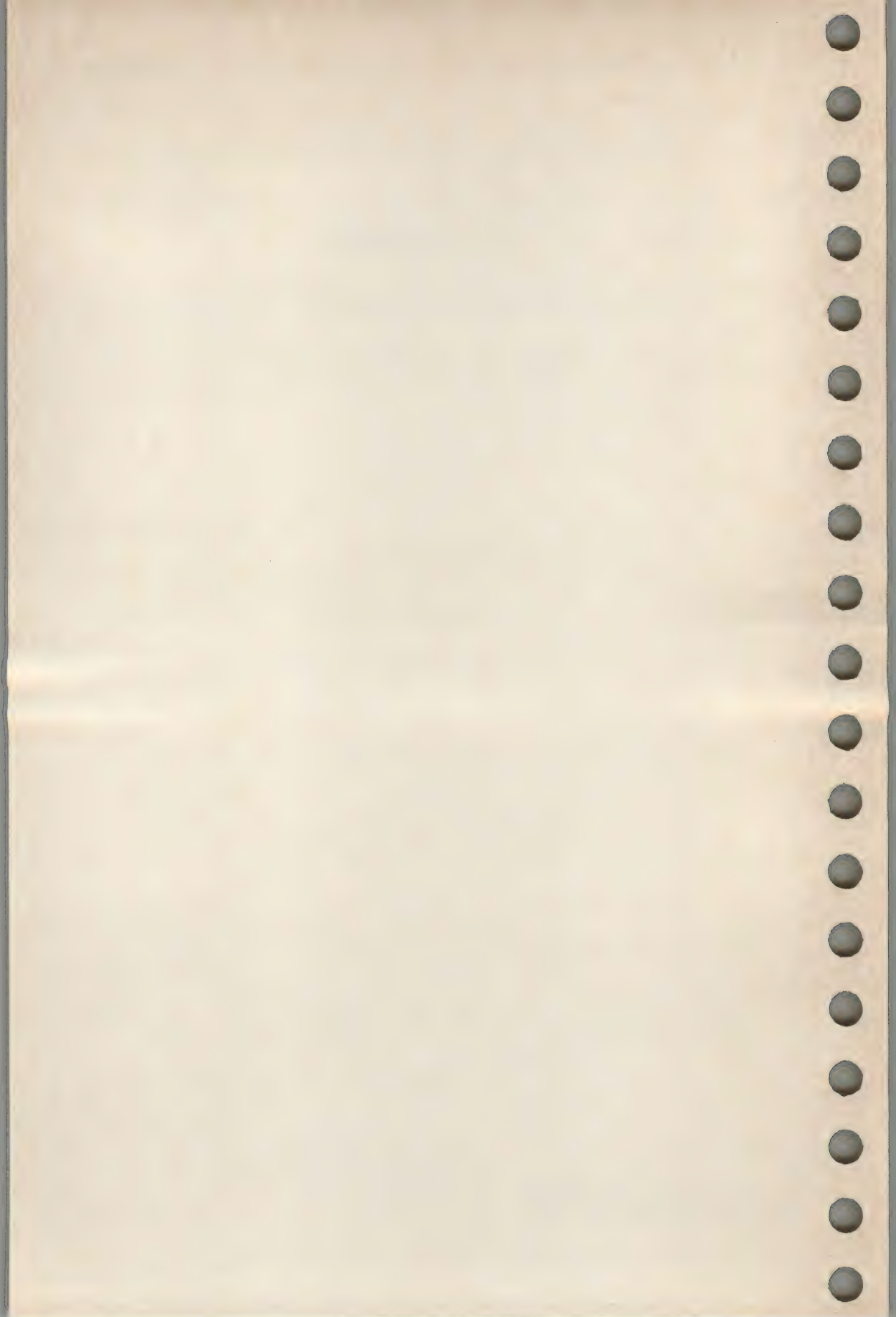
The explanation of what *stty 0* does has been expanded.

The obsolete *—lfkc* has been omitted.

The fact that *cs8* and no parity are used has been added to the description of the *raw* flag.

The values for the default ERASE and KILL characters have been marked as "system dependent".





## NAME

su — become super-user or another user

## SYNOPSIS

su [—] [name[ arg ...]]

## DESCRIPTION

The command *su* allows one to become another user without logging off. The default is to change to the super-user.

To use *su*, the appropriate password must be supplied (unless one is already the super-user). If the password is correct, *su* will execute a new environment with the real and effective user ID set to that of the specified user. The new command interpreter will be the optional program named in the specified user's password file entry, or the default if none is specified. Normal user ID privileges can be restored by entering EOF.

Any additional arguments given on the command line are passed to the command interpreter.

UN The following statements are true only if the command interpreter named in the specified user's password file entry is *sh*, see *sh*(1). If the first argument to *su* is a —, the environment will be changed to what would be expected if the user actually logged in as the specified user. Otherwise, the environment is passed along with the possible exception of PATH, which is set to a "safe" value if, and only if, the new user is the super-user.

## FILES

/etc/passwd	system's password file
/etc/profile	system's profile
\$HOME/.profile	user's profile

## SEE ALSO

*sh*(1).

## APPLICATION USAGE

The user should note that the real user ID is unchanged, so the login name from *who*, see *who*(1), will be unaffected.

## CHANGE HISTORY

First released in Issue 2.

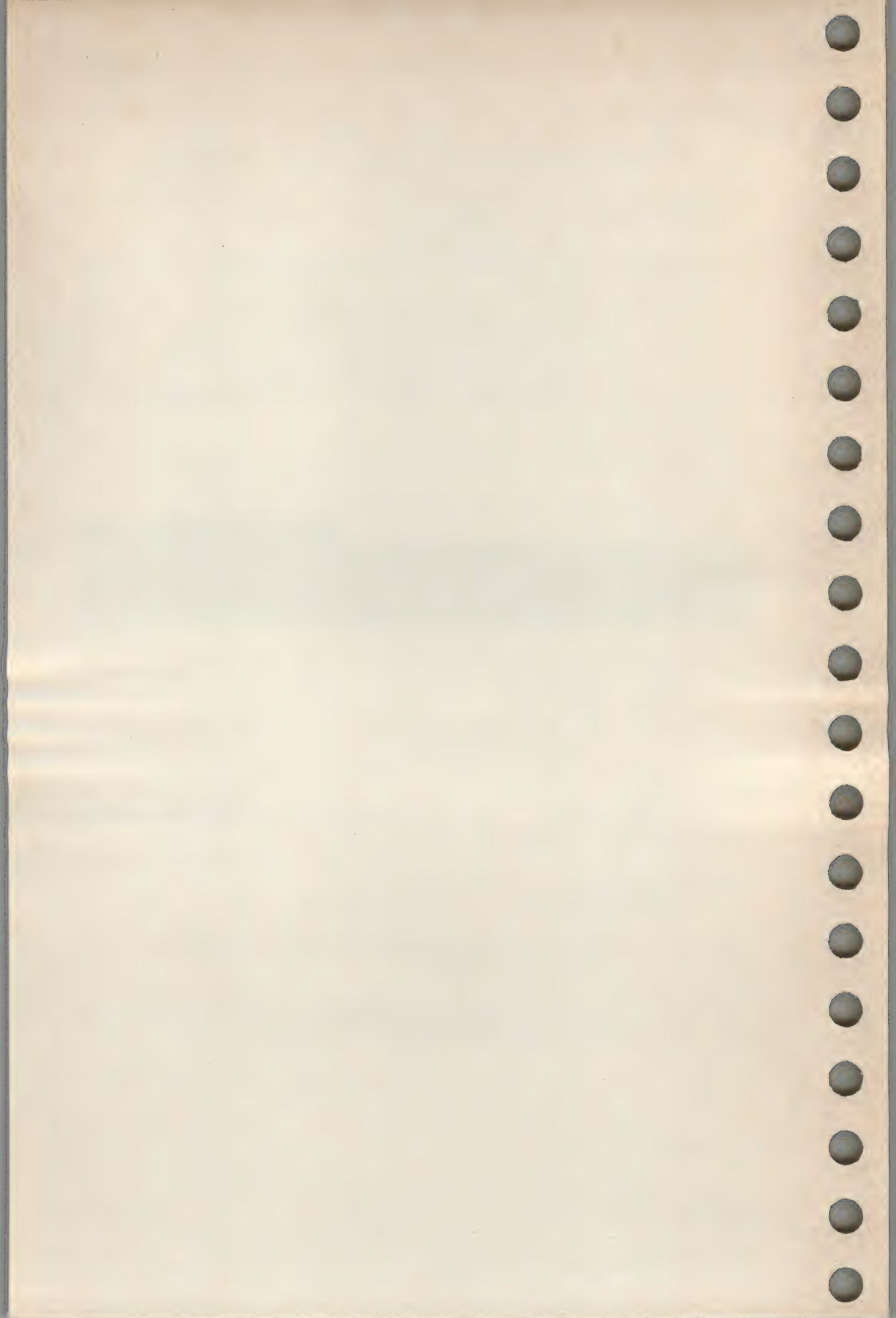
## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

A note about logging has been removed.

The words " , which is set to a "safe" value if, and only if, the new user is the super-user" have been added to the end of the final paragraph of the DESCRIPTION.





## NAME

sum — print checksum and block count of a file

## SYNOPSIS

sum [-r] [file...]

OF PI

## DESCRIPTION

The command *sum* calculates and prints a checksum for the named file, and also prints the space used by the file, in 512-byte units. The option *-r* causes an alternate algorithm to be used in computing the checksum. If no files are named, the standard input is read.

## APPLICATION USAGE

It is not clear that the algorithms used in typical implementations are portable, i.e., the same checksum is provided for the same input on different systems.

## CHANGE HISTORY

First released in Issue 2.

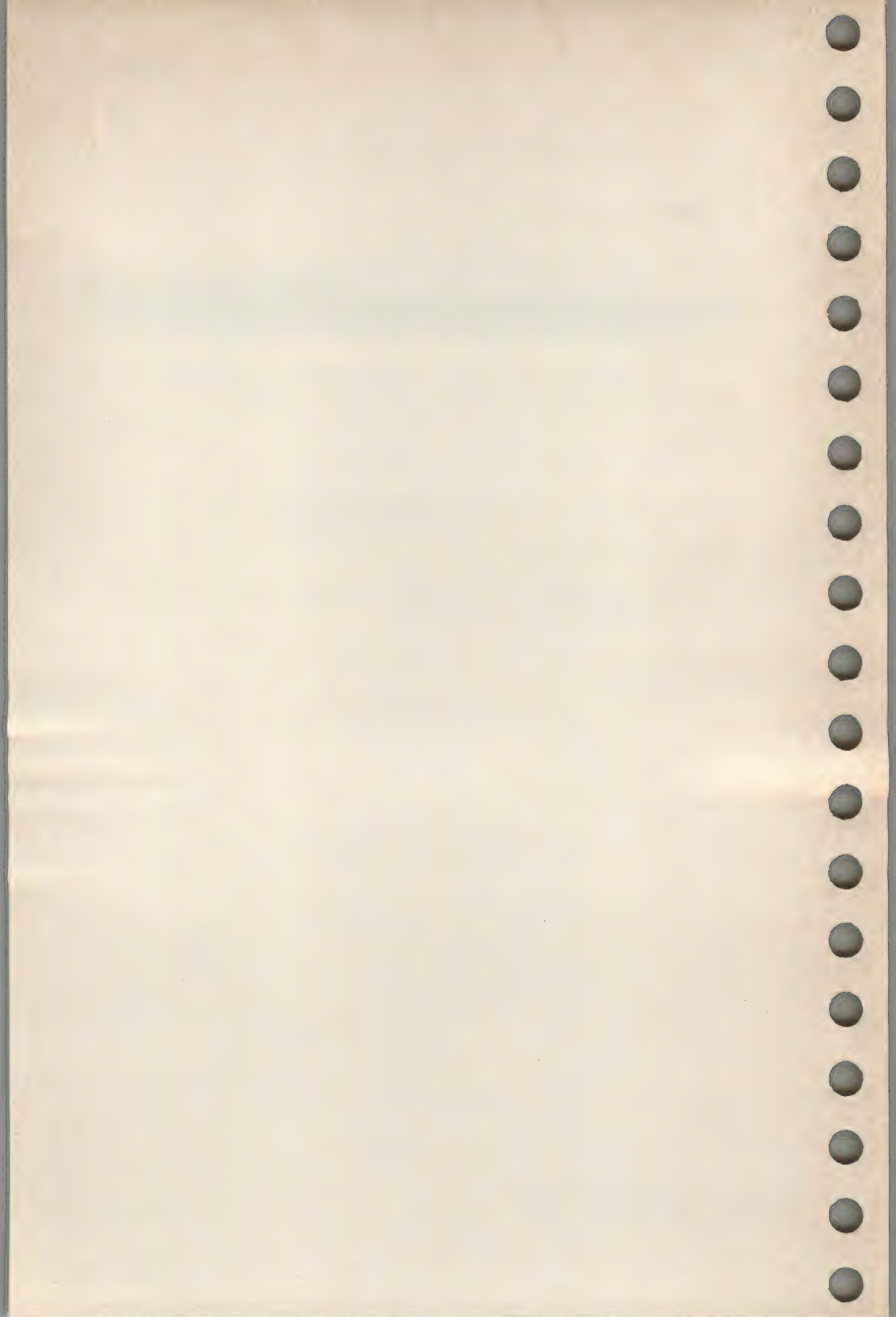
## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The paragraph indicating that the algorithms used are uniform across all System V implementations has been removed.

The SYNOPSIS and DESCRIPTION have been modified to indicate that the standard input or more than one file can be processed.





## NAME

`tabs` — set tabs on a terminal

## SYNOPSIS

`tabs [ tabspec ] [ +mn ] [ -Ttype ]`

MV UN

## DESCRIPTION

The command `tabs` sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings.

Three types of tab specification are accepted for *tabspec*: "canned," repetitive, arbitrary. If no *tabspec* is given, the default value is `-8`, i.e., "standard" tabs. The lowest column number is 1. Note that for `tabs`, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0.

## —code

Gives the name of one of a set of "canned" tabs. The legal codes and their meanings are as follows:

- a 1,10,16,36,72  
Assembler, applicable to some mainframes.
- a2 1,10,16,40,72  
Assembler, applicable to some mainframes.
- c 1,8,12,16,20,55  
COBOL, normal format
- c2 1,6,10,14,49  
COBOL compact format (columns 1-6 omitted).
- c3 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67  
COBOL compact format (columns 1-6 omitted), with more tabs than —c2.
- f 1,7,11,15,19,23  
FORTRAN
- p 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61  
PL/I
- s 1,10,55  
SNOBOL
- u 1,12,20,44  
Assembler, applicable to some mainframes.

In addition to these "canned" formats, three other types exist:

- n A repetitive specification requests tabs at columns  $1+n$ ,  $1+2*n$ , etc. Of particular importance is the value `-8`: this represents the "standard" tab setting, and is the most likely tab setting to be found at a terminal. Another special case is the value `-0`, implying no tabs at all.



*n1,n2,...*

The arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered identical.

Any of the following may be used also; if a given flag occurs more than once, the last value given takes effect:

—*Ttype*

The command *tabs* usually needs to know the type of terminal in order to set tabs and always needs to know the type to set margins. The argument *type* is a terminal name. If no —*T* flag is supplied, *tabs* searches for the environment variable TERM. If no *type* can be found, *tabs* tries a sequence that will work for many terminals.

+*mn* The margin argument may be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n+1* the left margin. If +*m* is given without a value of *n*, the value assumed is 10. The normal (leftmost) margin on most terminals is obtained by +*m0*. The margin for most terminals is reset only when the +*m* flag is given explicitly.

Tab and margin setting is performed via the standard output.

SEE ALSO

*stty*(1).

APPLICATION USAGE

This makes use of the terminal's hardware tabs, and also the *stty tabs* option.

This utility is not recommended for application use. It is a strong candidate for future removal.

CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

Unacknowledged trademarks have been deleted.

**NAME**

`tail` — deliver the last part of a file

**SYNOPSIS**

`tail [ ±[number][lbc[f]] [file]`

**DESCRIPTION**

The command `tail` copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance `+number` from the beginning, or `-number` from the end of the input (if *number* is null, the value 10 is assumed). The argument *number* is counted in units of lines, 512-byte units, or characters, according to the appended option *l*, *b*, or *c*. When no units are specified, counting is by lines.

With the `-f` (follow) option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop: it sleeps for at least a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail -f fred
```

will print the last ten lines of the file *fred*, followed by any lines that are appended to *fred* between the time `tail` is initiated and killed. As another example, the command:

```
tail -15cf fred
```

will print the last 15 characters of the file *fred*, followed by any lines that are appended to *fred* between the time `tail` is initiated and killed.

**APPLICATION USAGE**

Tails relative to the end of the file are saved in a buffer, and thus are limited in length.

Various kinds of anomalous behaviour may occur with character special files.

**CHANGE HISTORY**

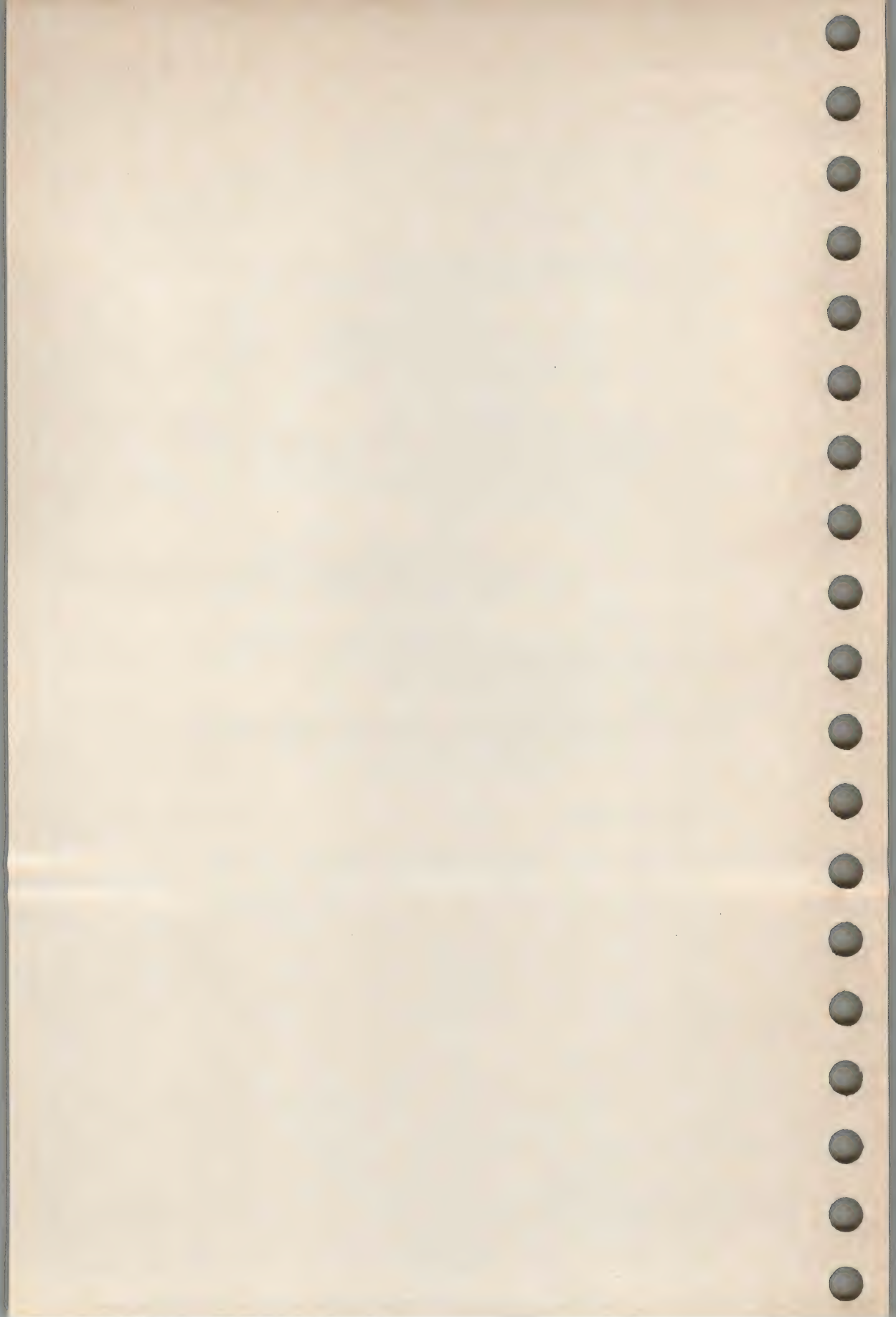
First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID with the following change:

The words "at least" have been added in the third paragraph of the **DESCRIPTION**.





## NAME

tar — file archiver

## SYNOPSIS

tar [ option ] [ file ... ]

## DESCRIPTION

The command *tar* creates archives of files. Its actions are controlled by the *option* argument. The *option* is a string of characters containing at most one function letter and possibly one or more modifiers. Other arguments to the command are *file* or files (or directory names) specifying which files are to be archived or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the option is specified by one of the following letters:

- r    The named *file* or files are written on the end of the archive.
- x    The named *file* or files are extracted from the archive. If a named file matches a directory whose contents had been written onto the archive, this directory is (recursively) extracted. If a named file in the archive does not exist on the system, the file is created with the same mode as the one in the archive, except that the set-user-ID and set-group-ID modes are not set unless the user is super-user. If the files exist, their modes are not changed except as described above. The owner, group, and modification time are restored (if possible). If no *files* argument is given, the entire content of the archive is extracted. Note that if several files with the same name are in the archive, the last one overwrites all earlier ones.
- t    The names of all the files in the archive are listed.
- u    The named *file* or files are added to the archive if they are not already there, or have been modified since last written into the archive.
- c    Create a new archive; writing begins at the beginning of the archive, instead of after the last file.

The following characters may be used in addition to the letter that selects the desired function:

- v    Normally, *tar* does its work silently. The *v* (verbose) modifier causes it to type the name of each file it treats, preceded by the option letter. With the *t* option, *v* gives more information about the archive entries than just the name.
- w    Causes *tar* to print the action to be taken, followed by the name of the file, and then wait for the user's confirmation. If a word beginning with *y* is given, the action is performed. Any other input means "no". This modifier is invalid with the *t* option.



- f** Causes *tar* to use the next argument as the name of the archive instead of the system dependent default. If the name of the file is `—`, *tar* writes to the standard output or reads from the standard input, whichever is appropriate. Thus, *tar* can be used as the head or tail of a pipeline. The command *tar* can also be used to move directory hierarchies with the command:
- ```
(cd fromdir; tar cf — .) | (cd todir; tar xf —)
```
- b** Causes *tar* to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 20. This option should only be used with (raw) magnetic tape archives (see *f* above). The block size is determined automatically when reading tapes (options *x* and *t*).
- l** Tells *tar* to report if it cannot resolve all of the links to the files being archived. If *l* is not specified, no error messages are printed. This modifier is valid only with the options *c*, *r*, and *u*.
- m** Tells *tar* not to restore the modification times. The modification time of the file will be the time of extraction. This modifier is invalid with the *t* option.
- PI** **o** Causes extracted files to take on the user and group identifier of the user running the program rather than those on the archive. This modifier is valid only with the *x* option.

#### EXAMPLE

Recursively write all files in this directory and below to tape, with blocking factor of 20, listing their names as written:

```
tar crvbf 20 /dev/rsctmtm0 .
```

#### ERRORS

The command *tar* reports bad option characters and read/write errors. It also reports an error if enough memory is not available to hold the link tables.

#### APPLICATION USAGE

"XVS SOURCE CODE TRANSFER" discusses use of *tar* between X/OPEN systems.

The *r* and *u* options may not function correctly if the archive being manipulated is a magnetic tape device.

Refer to "XVS SOURCE CODE TRANSFER" for guidelines on portability of source code between X/OPEN systems.

#### SEE ALSO

*cpio*(1).

CHANGE HISTORY

First released in Issue 2.

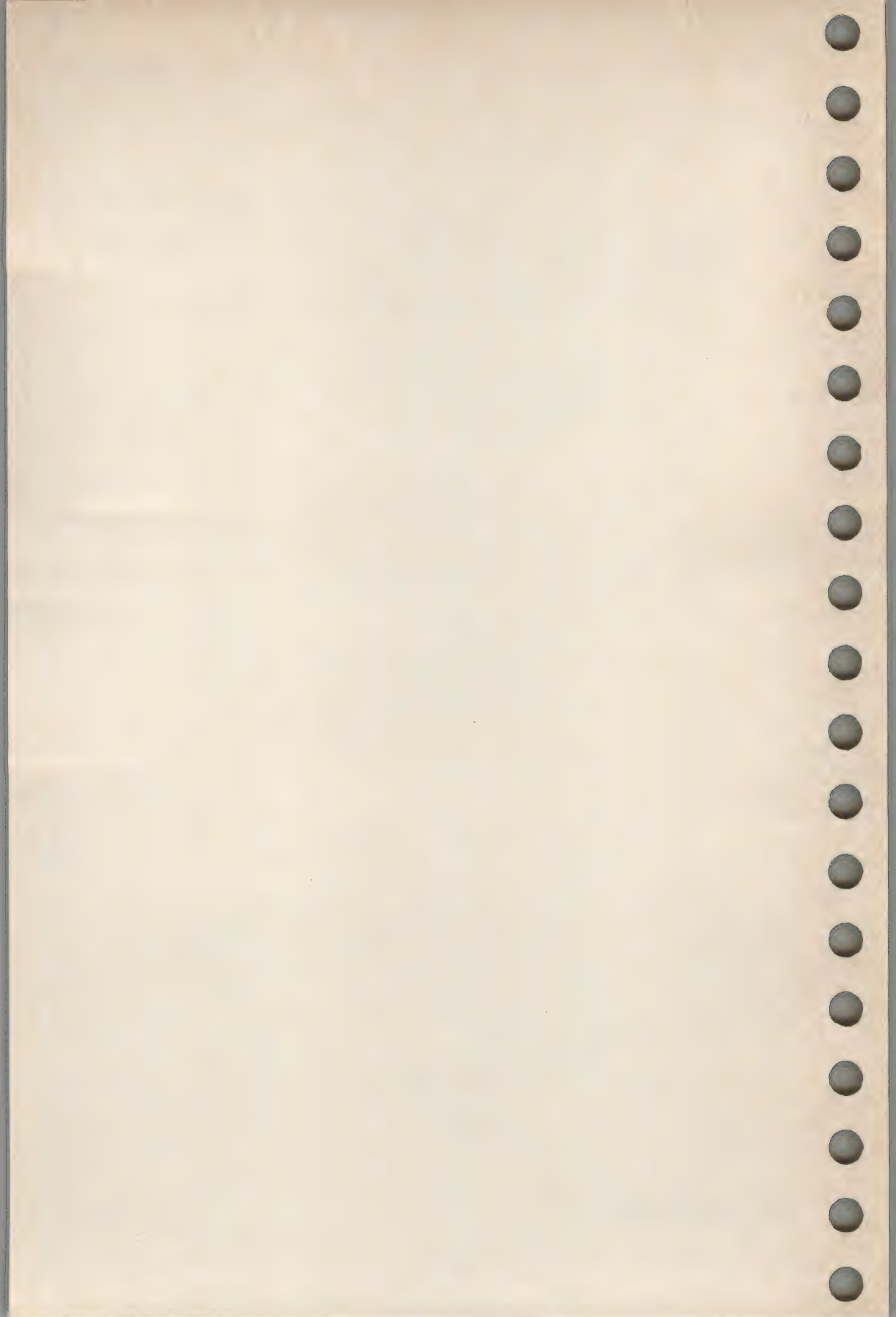
Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The sentence "This option implies the *r* option." has been deleted from the description of both the *c* and *u* options.

The words "system dependent" have been added to the description of the *f* option.





NAME

tee — join pipes and make copies of input

SYNOPSIS

tee [—i] [—a] [file...]

DESCRIPTION

The command *tee* transcribes the standard input to the standard output and makes copies in the *file* or files named. The *—i* option ignores interrupts; the *—a* option causes the output to be appended to the *file* or files rather than overwriting them.

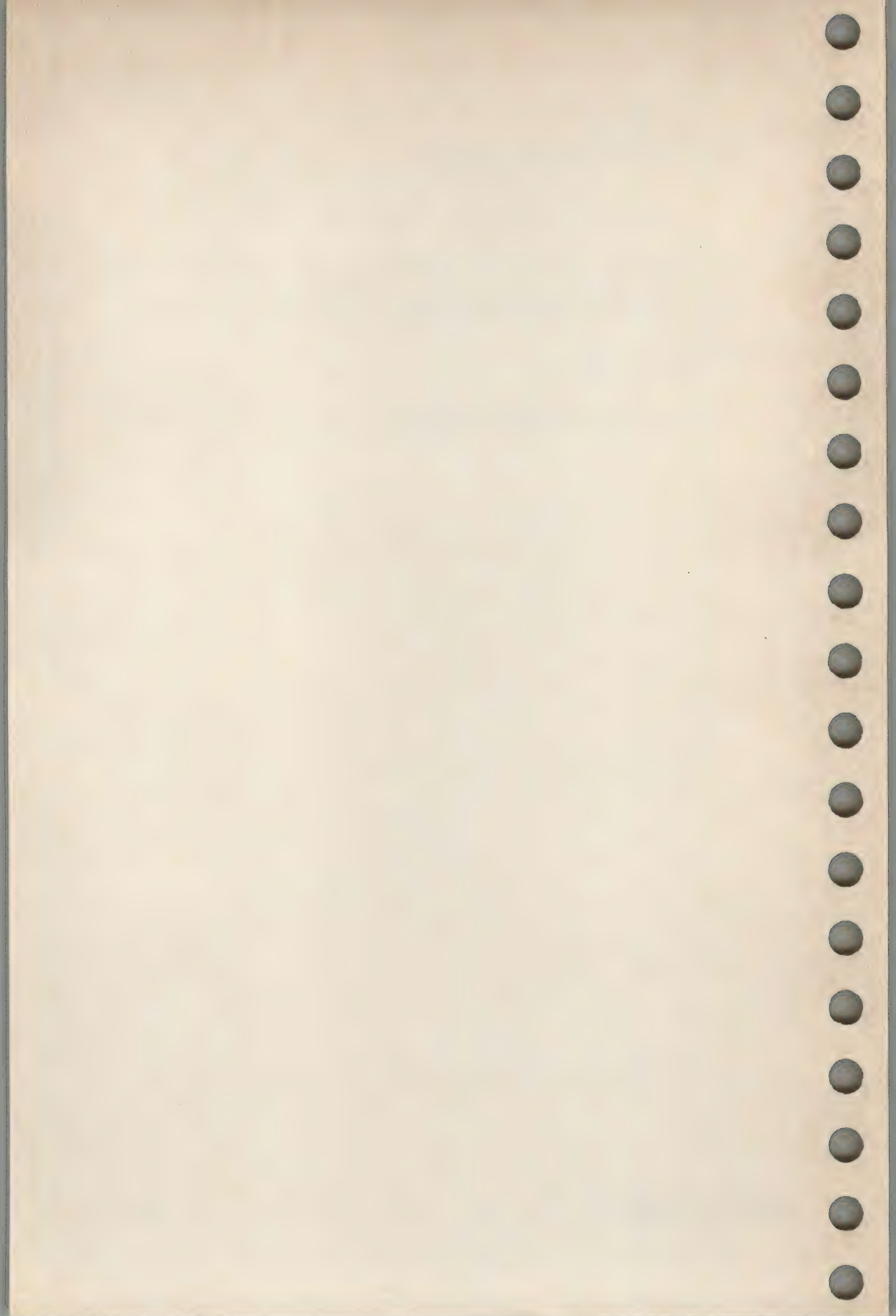
CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

test — condition evaluation command

## SYNOPSIS

test expr

[ expr ]

## DESCRIPTION

The command *test* evaluates the expression *expr* and, if its value is true, returns a zero (true) exit status; otherwise, a non-zero (false) exit status is returned; *test* also returns a non-zero exit status if there are no arguments. The following primitives are used to construct *expr*:

—r *file* true if *file* exists and is readable.

—w *file* true if *file* exists and is writable.

—x *file* true if *file* exists and is executable.

—f *file* true if *file* exists and is a regular file.

—d *file* true if *file* exists and is a directory.

—c *file* true if *file* exists and is a character special file.

—b *file* true if *file* exists and is a block special file.

UN

—p *file* true if *file* exists and is a named pipe (fifo)

—u *file* true if *file* exists and its set-user-ID bit is set.

—g *file* true if *file* exists and its set-group-ID bit is set.

—s *file* true if *file* exists and has a size greater than zero.

—t [*fildevs*]

true if the open file whose file descriptor number is *fildevs* (1 by default) is associated with a terminal device.

—z *s1* true if the length of string *s1* is zero.

—n *s1* true if the length of the string *s1* is non-zero.

*s1* = *s2* true if strings *s1* and *s2* are identical.

*s1* != *s2* true if strings *s1* and *s2* are not identical.

*s1* true if *s1* is not the null string.

*n1* —eq *n2*

true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons —ne, —gt, —ge, —lt, and —le may be used in place of —eq.



# TEST(1)

Utilities

These primaries may be combined with the following operators:

- ! unary negation operator.
- a binary *and* operator.
- o binary *or* operator (—a has higher precedence than —o).
- ( *expr* ) parentheses for grouping.

Notice that all the operators and flags are separate arguments to *test*. Notice also that parentheses may be meaningful to the command interpreter and therefore may need to be escaped.

In the second form of the command (i.e., the one that uses *[]*, rather than the word *test*), the square brackets must be delimited by blanks.

## SEE ALSO

find(1), sh(1).

## APPLICATION USAGE

This is usually a shell built-in command.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.

NAME

time — time a command

SYNOPSIS

time command [ arg ... ]

DESCRIPTION

The *command* is executed; after it is complete, *time* prints the elapsed time during the command, the time spent executing system code, and the time spent in execution of the user code. Times are reported in seconds.

The times are printed on standard error.

Optionally, *arguments* may be passed to the command by placing them as separate words after the command name.

APPLICATION USAGE

When *time* is used on a multi-processor system the sum of system and user time could be greater than real time.

CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The paragraph about passing arguments to the command has been added.





## NAME

`touch` — update access and modification times of a file

## SYNOPSIS

`touch [—amc] [mmddhhmm[yy]] file...`

## DESCRIPTION

The command *touch* causes the access and modification times of each *file* to be updated. The *file* is created if it does not exist. The `—a` and `—m` options cause *touch* to update only the access or modification times respectively (default is `—am`). The `—c` option silently prevents *touch* from creating the *file* if it did not previously exist.

UN

If a time is specified, that time is used in place of the current time

## EXIT STATUS

The exit status from *touch* is the number of files for which the times could not be successfully modified (including files that did not exist and were not created).

## APPLICATION USAGE

Completely numeric filenames may cause confusion as *touch* may assume this is a date argument.

## FUTURE DIRECTIONS

Because of the parsing ambiguity introduced by the date format marked as UN, a `—d` date option will be introduced to support this functionality.

## CHANGE HISTORY

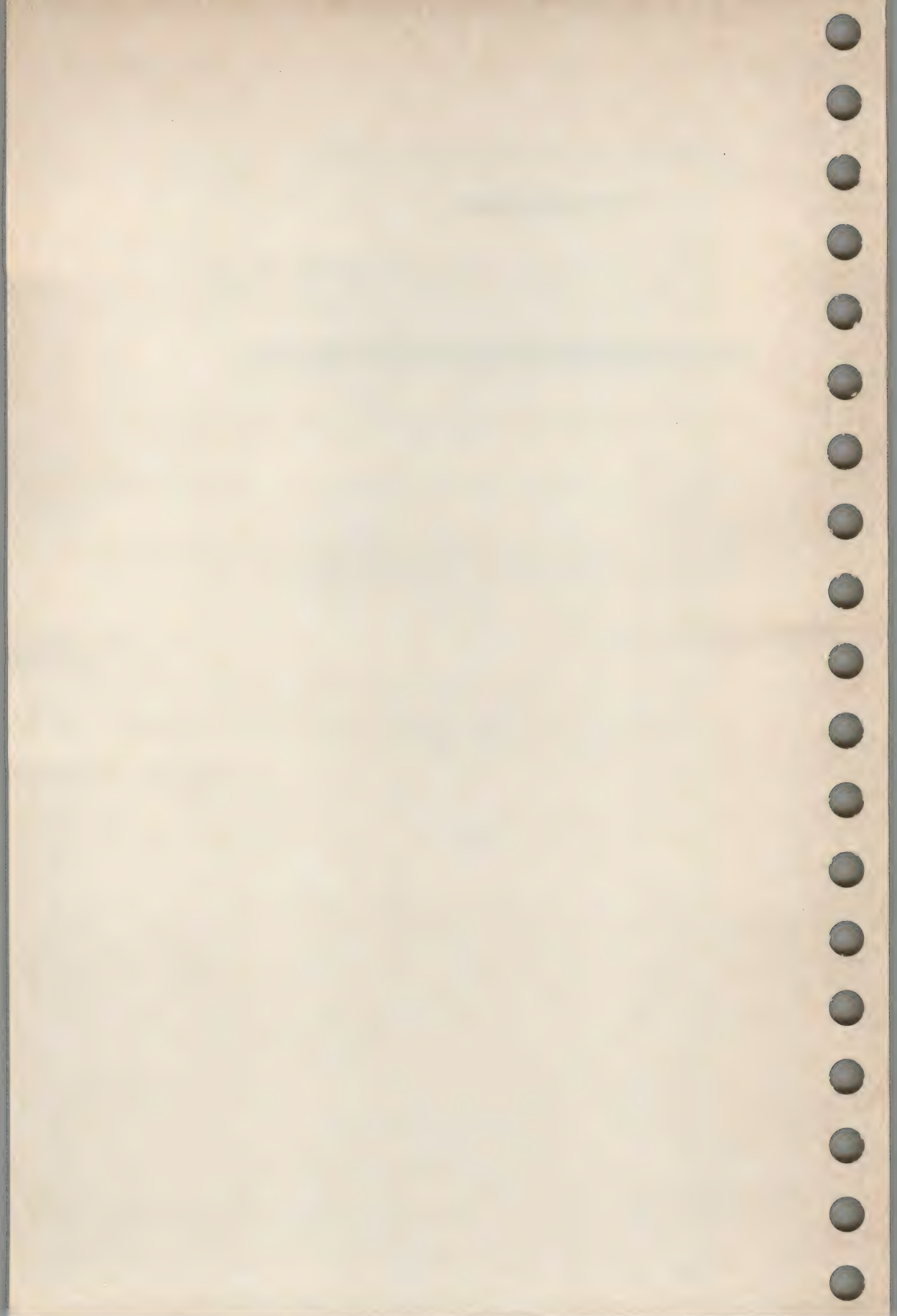
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The words "If no time is specified the current time is used" have been replaced by "If a time is specified, that time is used in place of the current time".





## NAME

*tr* — translate characters

## SYNOPSIS

*tr* [*—cds*] [*string1* [*string2*]]

## DESCRIPTION

The command *tr* copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. Any combination of the options *—cds* may be used:

- c* Complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 001 through 377 octal.
- d* Deletes all input characters in *string1*. *String2* is ignored.
- s* Squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

- [*a-z*] Stands for the string of characters whose ASCII codes run from character *a* to character *z*, inclusive.
- [*a\*n*] Stands for *n* repetitions of *a*. If the first digit of *n* is 0, *n* is considered octal; otherwise, *n* is taken to be decimal. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character *\* may be used to remove special meaning from any character following it in a string. In addition, *\* followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

## EXAMPLE

The following example creates a list of all the words in *file1* one per line in *file2*, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the command interpreter; 012 is the ASCII code for newline.

```
tr —cs "[A-Z][a-z]" "[\012*]" <file1 >file2
```

## APPLICATION USAGE

The command *tr* does not handle ASCII NUL in *string1* or *string2*; it always deletes NUL from input.

## CHANGE HISTORY

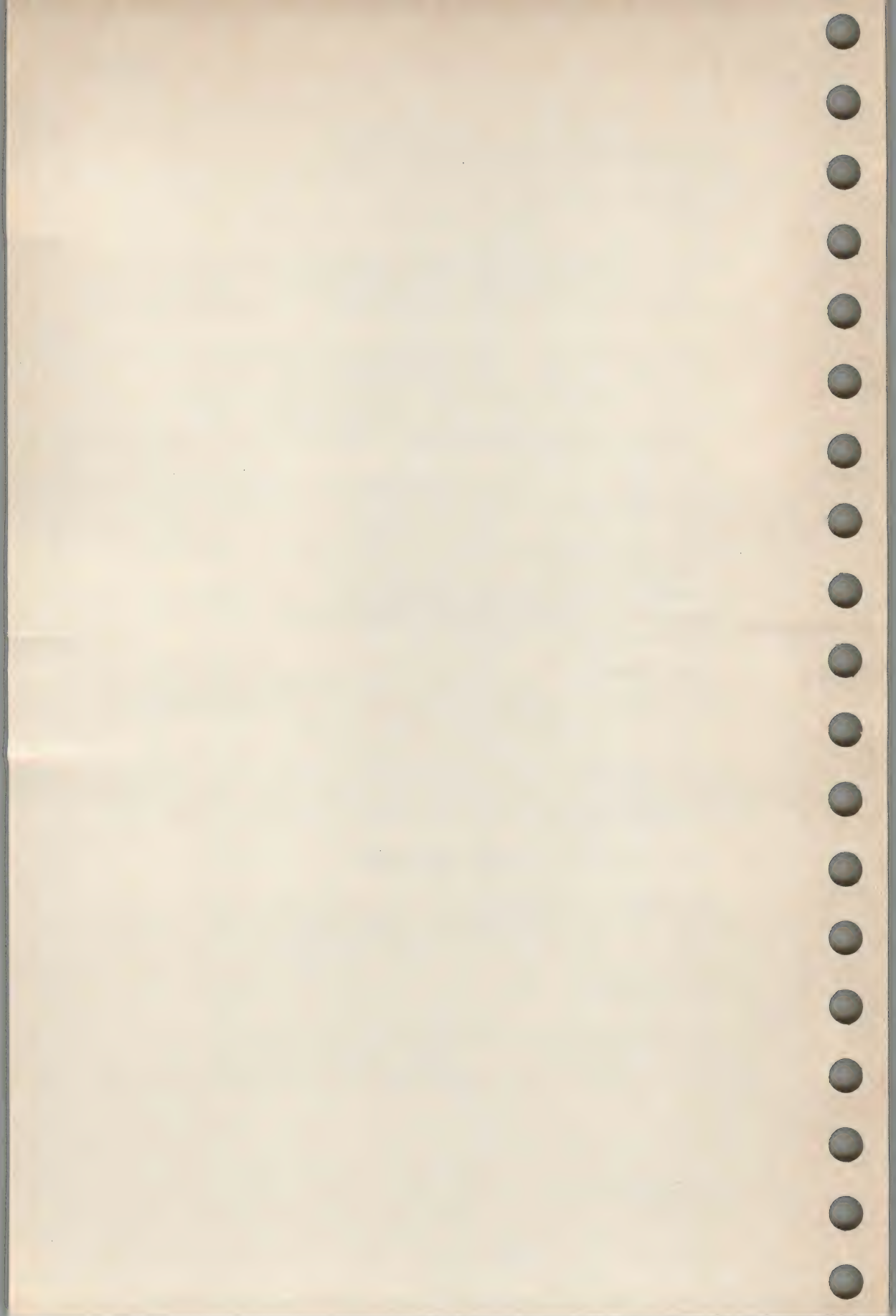
First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The sentence "*String2* is ignored." has been added to the description of the *—d* option.





## NAME

true, false — provide truth values

## SYNOPSIS

true

false

## DESCRIPTION

The command *true* does nothing, and returns exit code zero. The command *false* does nothing, and returns a non-zero exit code. They are typically used to construct command procedures.

## EXAMPLE

This command will be executed forever:

```
while true
do
    command
done
```

## EXIT STATUS

Exit status is:

|              |          |
|--------------|----------|
| <i>true</i>  | zero     |
| <i>false</i> | non-zero |

## SEE ALSO

sh(1).

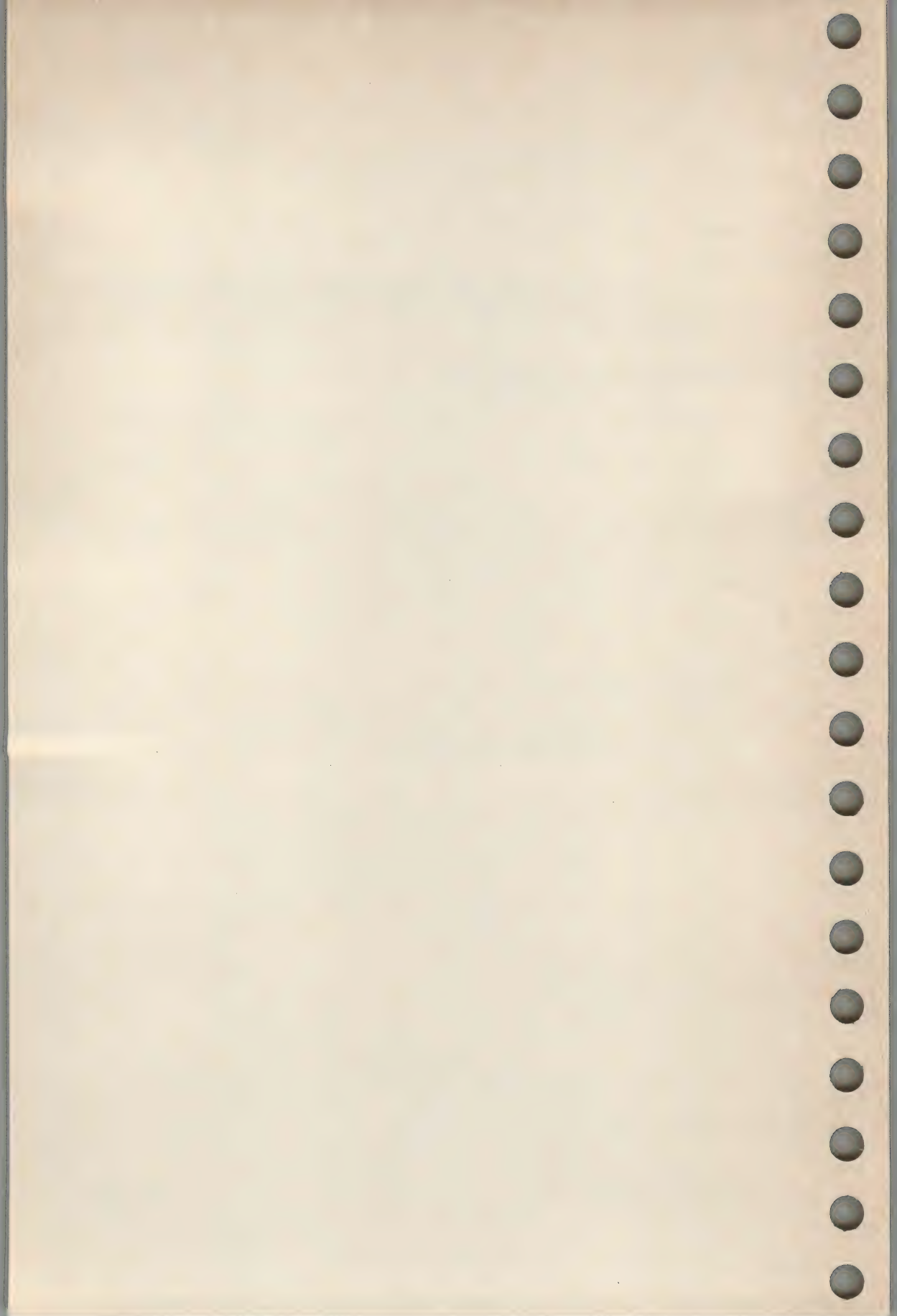
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





NAME

tsort — topological sort

SYNOPSIS

tsort [ file ]

DESCRIPTION

*Tsort* produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (non-empty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1D).

APPLICATION USAGE

*Tsort* is typically used in conjunction with *lorder*, see *lorder*(1D).

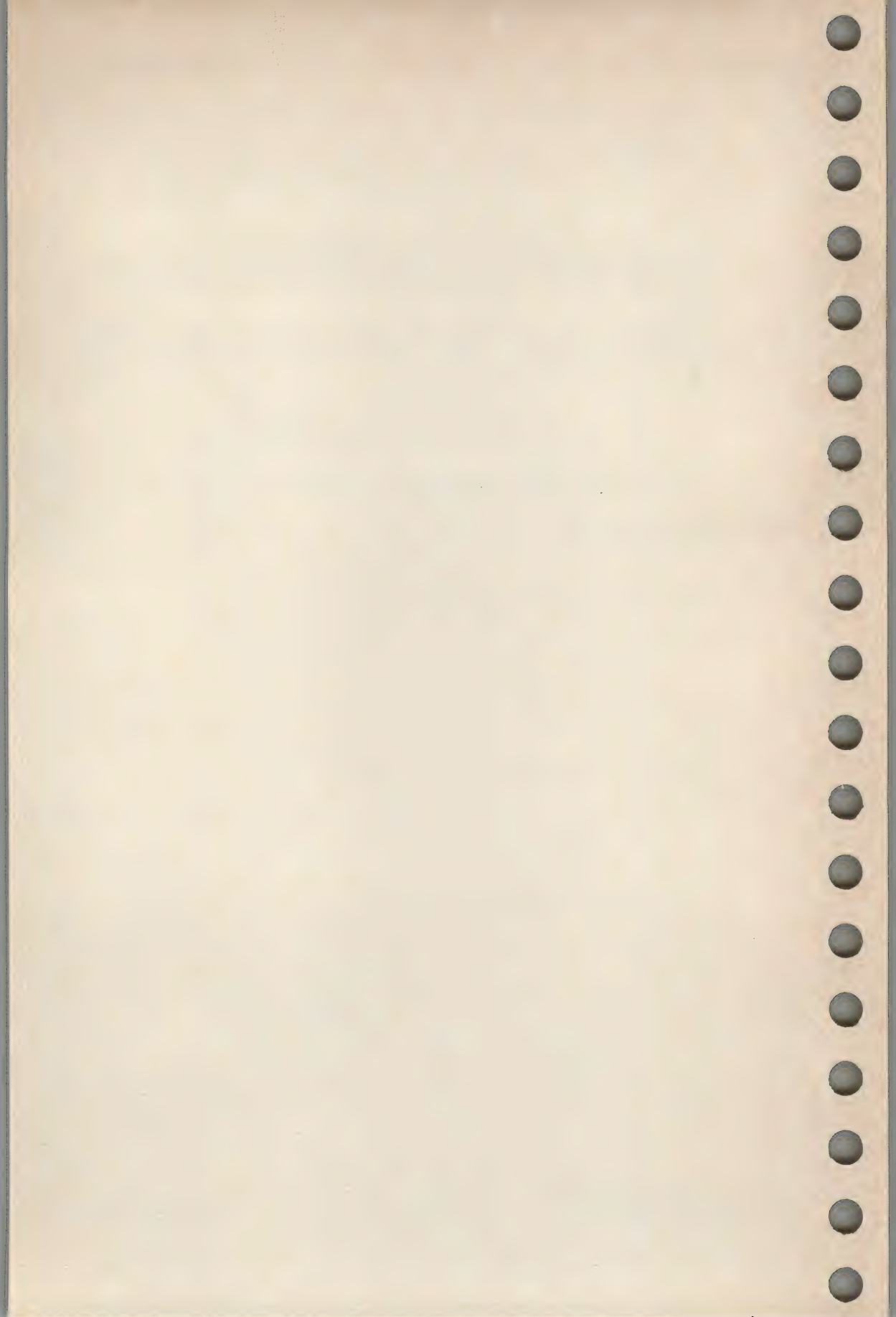
CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID.





NAME

tty — get the name of the terminal

SYNOPSIS

tty [—s]

DESCRIPTION

The command *tty* prints the path name of the user's terminal. The *—s* option inhibits printing of the terminal path name, allowing one to test just the exit code.

EXIT STATUS

Exit codes:

- 0 if standard input is a terminal,
- 1 otherwise,
- 2 if invalid options were specified.

An error is reported if the standard input is not a terminal and *—s* is not specified.

APPLICATION USAGE

On systems supporting virtual terminals, *tty* returns the name of the virtual terminal, not the real terminal, when the former is in use.

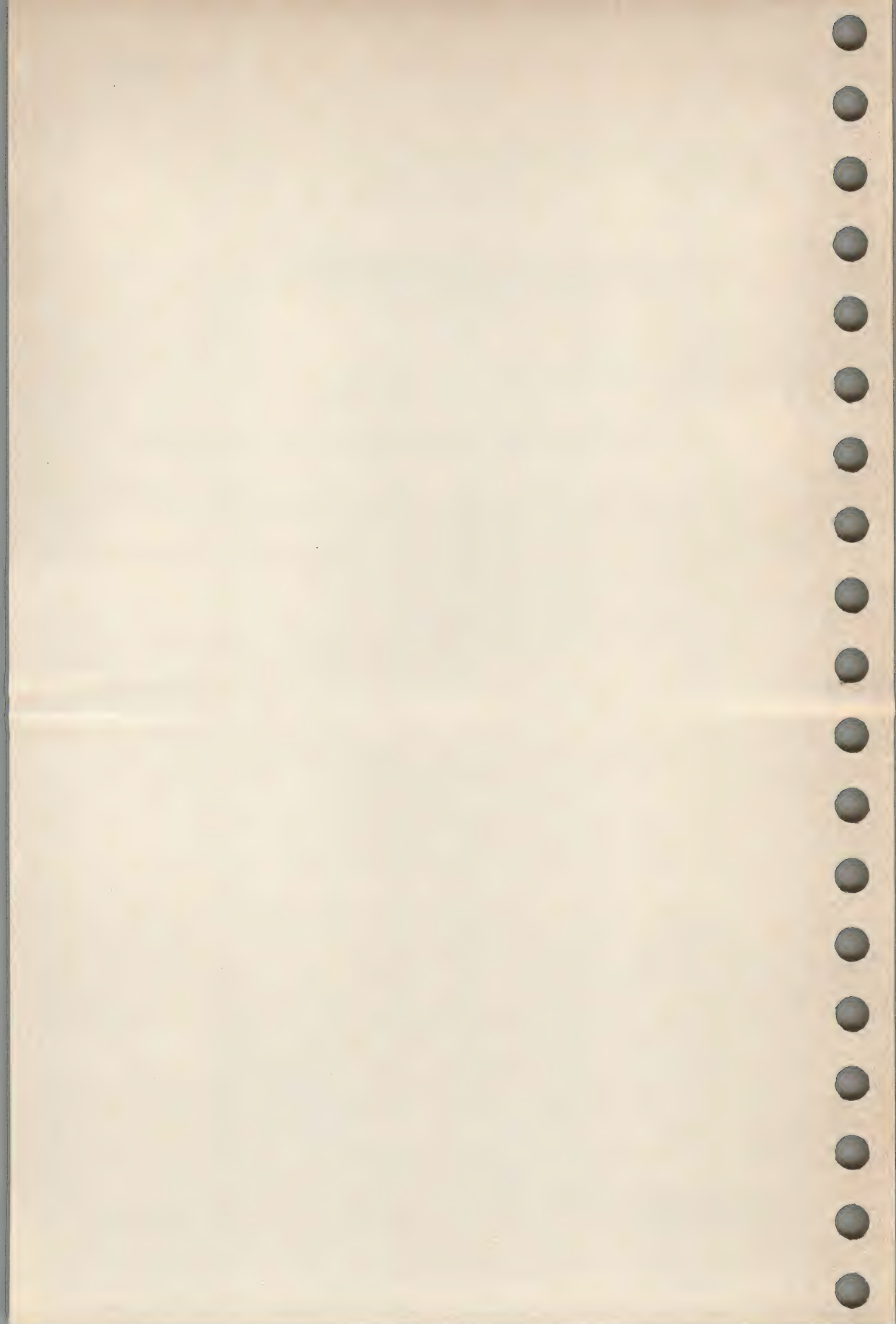
CHANGE HISTORY

First released in Issue 2.

Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

umask — set file-creation mode mask

## SYNOPSIS

umask [ooo]

## DESCRIPTION

The user file-creation mode mask is set to *ooo*. These three octal digits refer to read/write/execute permissions for *owner*, *group*, and *others*, respectively, see *chmod*(1). The bits which are set to 1 in *ooo* will be zero in the permission field of all subsequently created files. *ooo* are set to zero by the system for all subsequent file creations.

For example, *umask 022* removes *group* and *others* write permission (files normally created with mode 777 become mode 755; files created with mode 666 become mode 644).

If *ooo* is omitted, the current value of the mask is printed.

## SEE ALSO

*chmod*(1), *umask*(2).

## APPLICATION USAGE

This is always a shell built in command.

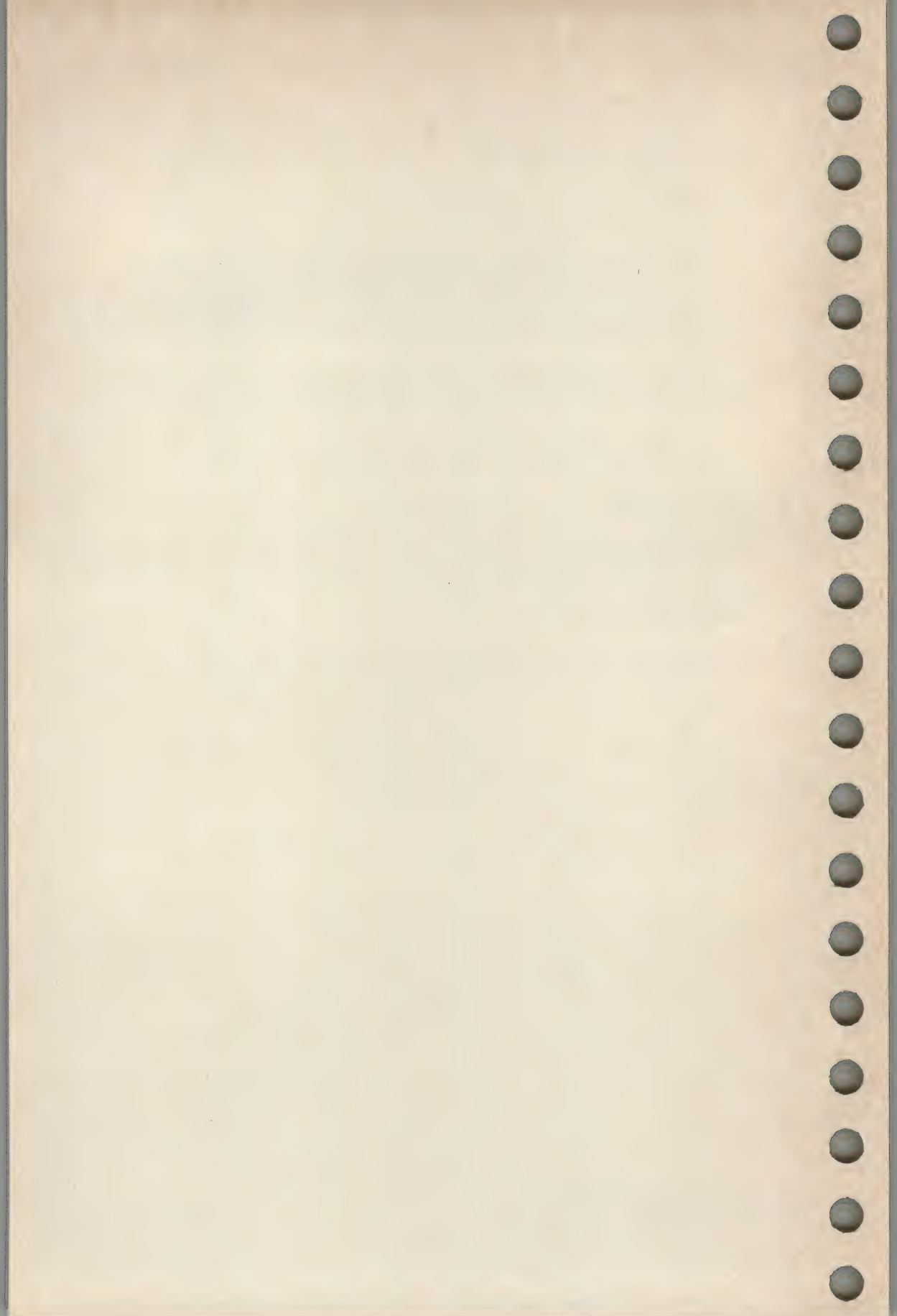
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

uname — print name of current system

## SYNOPSIS

uname [ —snrvma ]

## DESCRIPTION

The command *uname* prints the current system name on the standard output file. The options cause selected information returned by *uname*(2) to be printed:

- s print the system name (default). This is a name by which the system is known in the local installation.
- n print the nodename. The nodename may be a name by which the system is known to a communications network.
- r print the operating system release.
- v print the operating system version.
- m print the machine hardware name.

OF

—a print all the above information.

OF

Combinations of the above options may be used.

## SEE ALSO

uname(2).

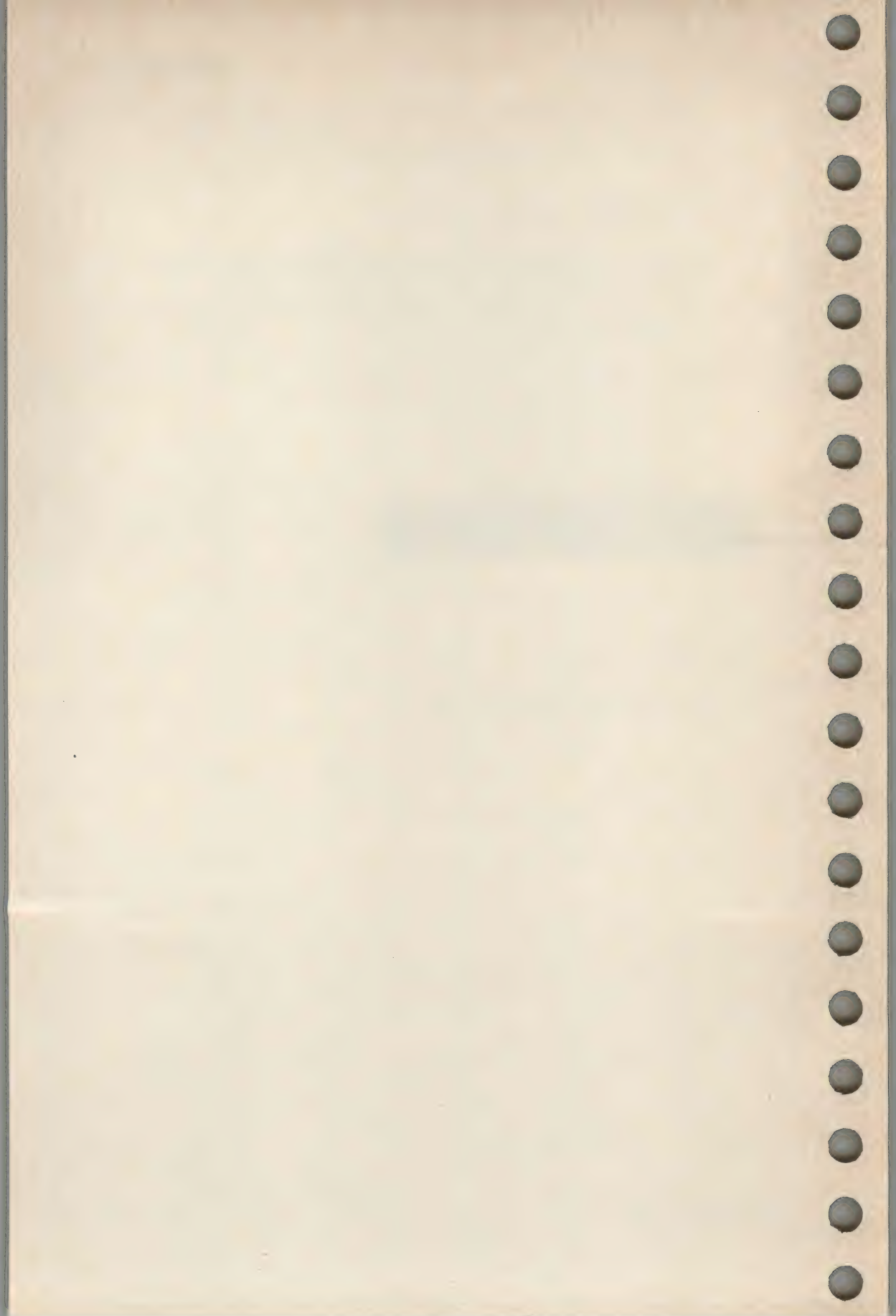
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

unget — undo a previous get of an SCCS file

## SYNOPSIS

unget [—rSID] [—s] [—n] files

## DESCRIPTION

*Unget* undoes the effect of a *get* —*e* done prior to creating the intended new delta. If a directory is named, *unget* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of — is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Keyletter arguments apply independently to each named file.

## —rSID

Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the new delta.) The use of this option is necessary only if two or more outstanding *gets* for editing on the same SCCS file were done by the same person (login name). An error is reported if the specified SID is ambiguous, or if it is necessary and omitted on the command line.

—s Suppresses the printout, on the standard output, of the intended delta's SID.

—n Causes the retention of the file that was obtained by *get*, which would normally be removed from the current directory.

## SEE ALSO

delta(1D), get(1D), sact(1D).

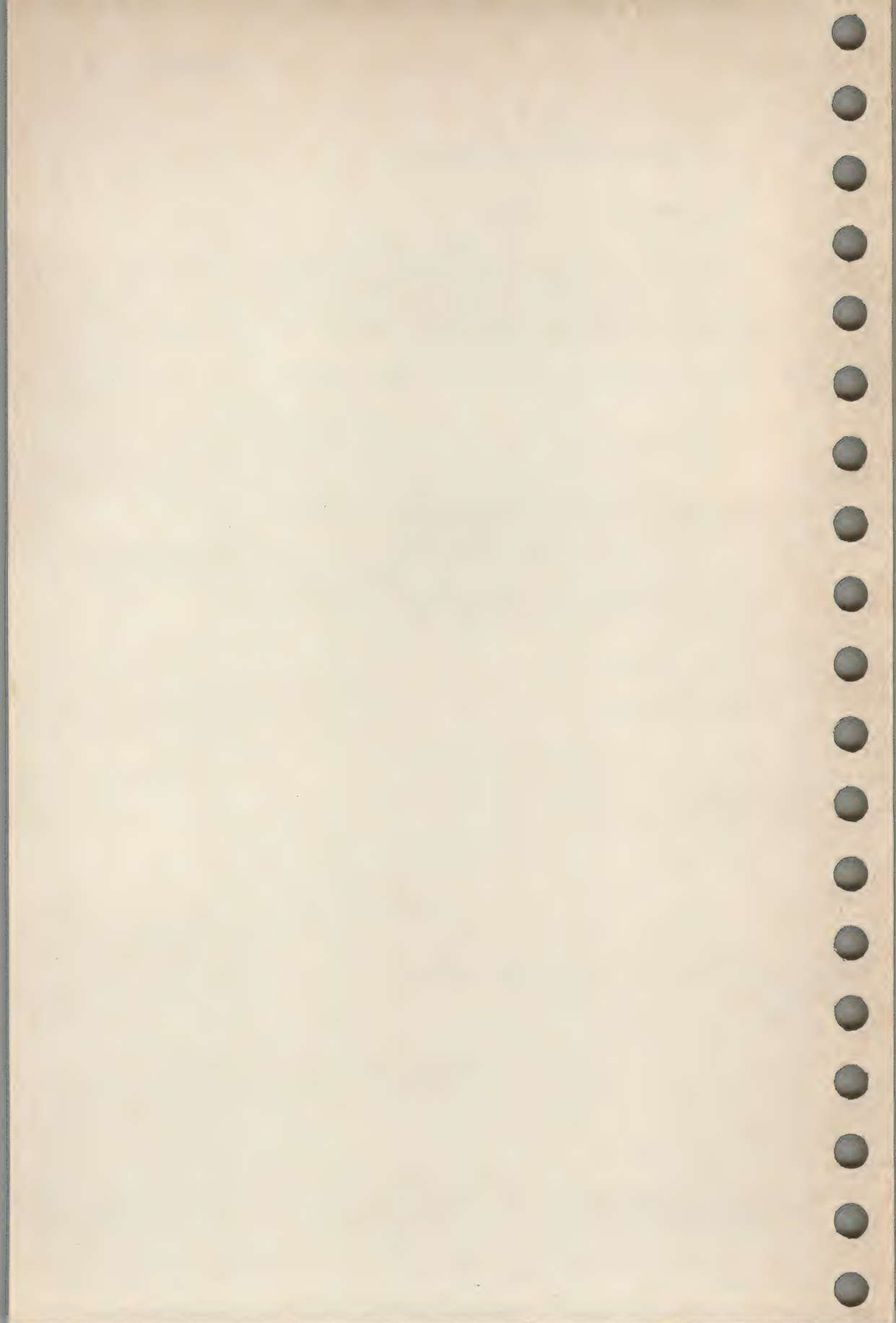
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

uniq — report repeated lines in a file

## SYNOPSIS

uniq [**—udc** [**+n**] [**—n**]] [*input* [*output*]]

## DESCRIPTION

The command *uniq* reads the *input* file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the *output* file. If *input* and *output* are omitted, *uniq* reads from standard input and writes to standard output. If *output* is omitted, standard output is assumed. The arguments *input* and *output* should always be different. Note that repeated lines must be adjacent in order to be found, see *sort*(1). If the **—u** flag is used, just the lines that are not repeated in the original file are output. The **—d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **—u** and **—d** mode outputs.

The **—c** option supersedes **—u** and **—d** and generates an output report with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

**—n**     The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbours.

**+n**     The first *n* characters are ignored. Fields are skipped before characters.

## SEE ALSO

comm(1), sort(1).

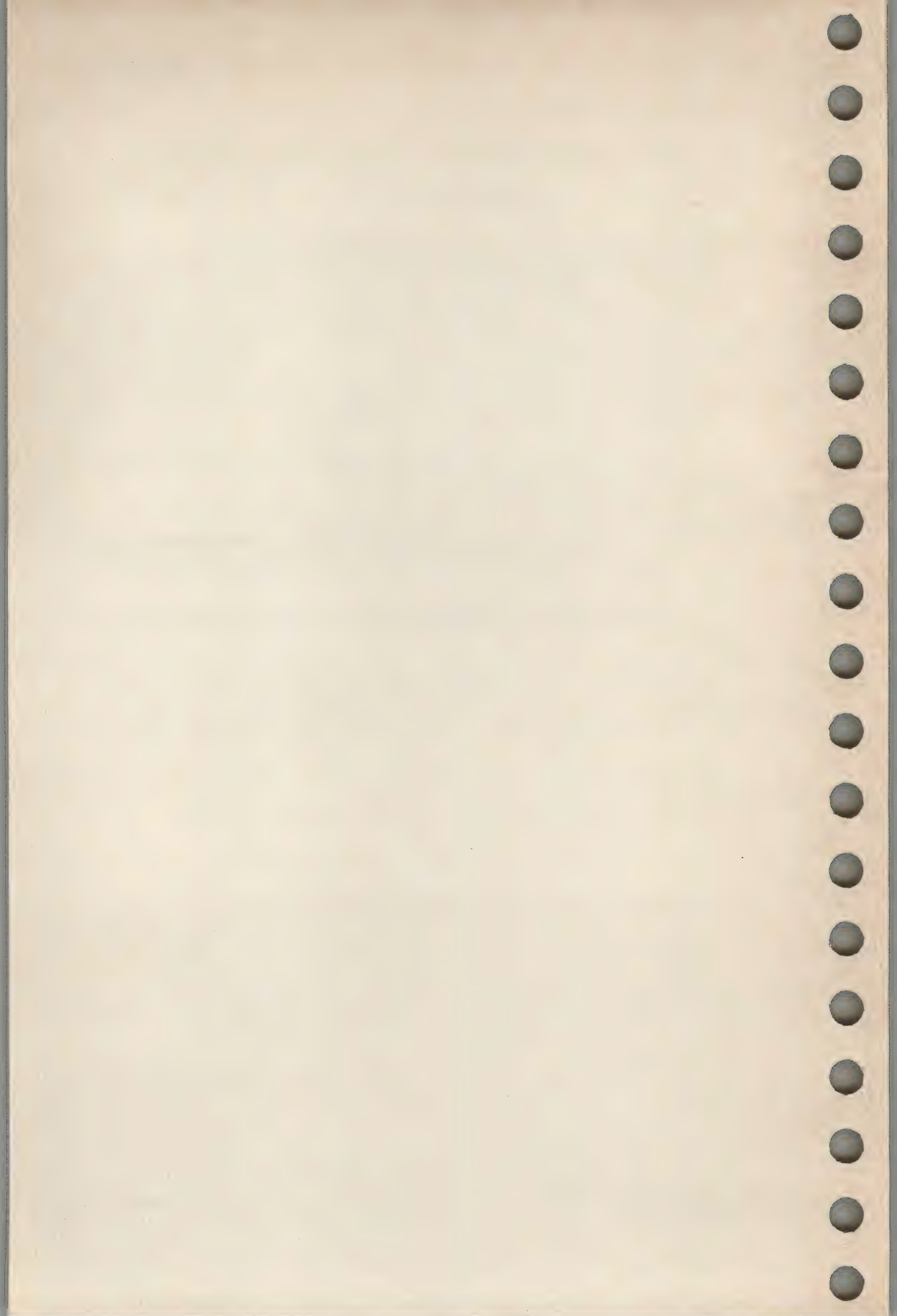
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

uucp, uulog, uuname — system-to-system copy

## SYNOPSIS

**uucp** [ options ] *source-file* ... *destination-file*

UN

**uulog** [ —s *system* ]

UN

**uuname** [ —l ]

UN

## DESCRIPTION

**uucp**

The command *uucp* copies files named by the *source-file* arguments to the *destination-file* argument. A file name may be a path name on your machine, or may have the form:

*system-name*!path-name

where *system-name* is taken from a list of system names that *uucp* knows about. The destination *system-name* may also be a list of names such as

*system-name*!system-name!...!system-name!path-name

in which case, an attempt is made to send the file via the specified route to the destination. Care should be taken to ensure that intermediate nodes in the route are willing to forward information.

The shell metacharacters *?*, *\**, and *[...]* appearing in *path-name* will be expanded on the appropriate system.

Path-names may be one of:

1. a full path-name.
2. a path-name preceded by *~user* where *user* is a login name on the specified system and is replaced by that user's login directory. Note that if an invalid login is specified, the default will be to the public directory (PUBDIR).
3. a path-name preceded by *~/destination* where *destination* is appended to PUBDIR

**Note:** This destination will be treated as a file name unless more than one file is being transferred by this request or the destination is already a directory. To ensure that it is a directory, follow the destination with a */*. For example, *~/dan/* as the destination will make the directory PUBDIR/*dan* if it does not exist and put the requested file(s) in that directory.

4. anything else is prefixed by the current directory.

If the result is an erroneous path-name for the remote system, the copy will fail. If the *destination-file* is a directory, the last part of the *source-file* name is used.

The command *uucp* gives universal read and write permissions and preserves execute permissions across the transmission.



The following options are interpreted by *uucp*:

- c Do not copy local file to the spool directory for transfer to the remote machine (default).
- UN —C Force the copy of local files to the spool directory for transfer
- d Make all necessary directories for the file copy (default).
- UN —f Do not make intermediate directories for the file copy.
- UN —j Output the job identification ASCII string on the standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.
- m Send mail to the requester when the copy is completed.
- UN —nuser Notify *user* on the remote system that a file was sent.
- UN —r Do not start the file transfer; just queue the job.

## uulog

The command *uulog* queries a log file of *uucp* or *uux* transactions.

- OF If the —s option is specified, then *uulog* prints information about file transfer work involving system *system*.

## uname

The command *uname* lists the *uucp* names of known systems, one per line. The —l option returns the local system name.

## SEE ALSO

mail(1), uustat(1), uux(1).

## APPLICATION USAGE

The domain of remotely accessible files can (and for obvious security reasons usually should) be severely restricted.

Typical implementations of this utility require a communications line configured to use the *termio*(7) interface. On systems where none of these lines are available, this utility may not be present.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following changes:

The reference to *uuxqt* transactions has been changed to *uux*.

The words "one per line" have been added to the description of the output of the *uname* command.

## NAME

`uustat` — `uucp` status inquiry and job control

## SYNOPSIS

`uustat [ options ]`

UN

## DESCRIPTION

The command `uustat` will display the status of, or cancel, previously specified `uucp` commands, or provide general status on `uucp` connections to other systems.

Not all combinations of *options* are valid. Only one of the following *options* can be specified with `uustat`:

UN

`—q` List the jobs queued for each machine.

`—k jobid`

Kill the `uucp` request whose job identification is *jobid*. The killed `uucp` request must belong to the person issuing the `uustat` command unless that user is the superuser.

UN

`—r jobid`

Rejuvenate *jobid*. The files associated with *jobid* are touched so that their modification time is set to the current time. This prevents the cleanup program from deleting the job until the jobs modification time reaches the limit imposed by the program.

The *options* below may not be used with the ones listed above; however, these *options* may be used singly or together:

`—s sys`

Report the status of all `uucp` requests for remote system *sys*.

`—u user`

Report the status of all `uucp` requests issued by *user*.

When no *options* are given, `uustat` outputs the status of all `uucp` requests issued by the current user.

## SEE ALSO

`uucp(1)`.

## APPLICATION USAGE

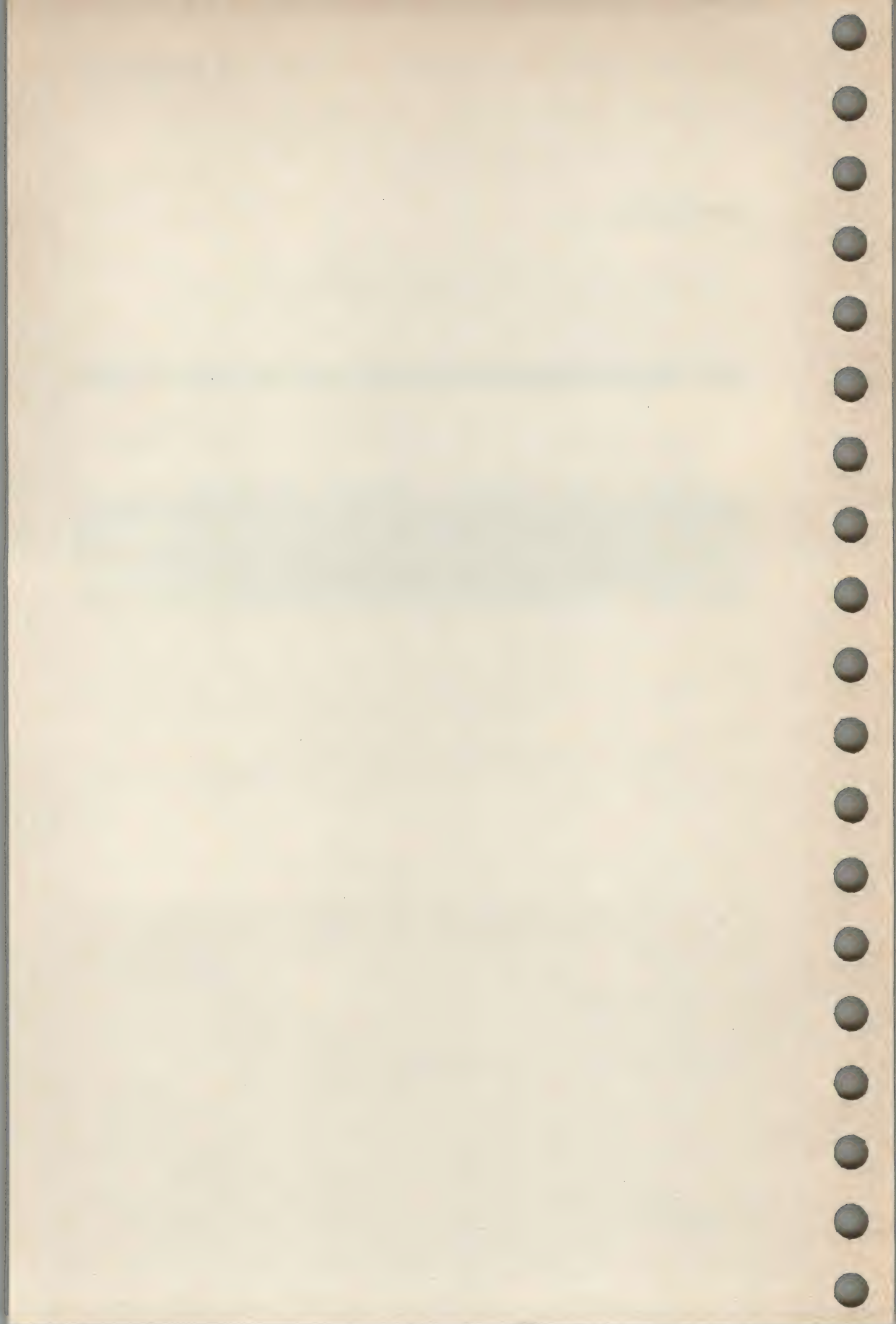
Typical implementations of this utility require a communications line configured to use the `termio(7)` interface. On systems where none of these lines are available, this utility may not be present.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

uuto, uupick — public system-to-system file copy

## SYNOPSIS

```
uuto [ -p ] [ -m ] source-file ... destination
```

UN

```
uupick [ -s system ]
```

UN

## DESCRIPTION

## uuto

The command *uuto* sends *source-files* to *destination*. The command *uuto* uses the *uucp*(1) facility to send files, while it allows the local system to control the file access. A *source-file* name is a path name on the user's machine. Destination has the form: *system!user* where *system* is taken from a list of system names that *uucp* knows about, see *uname* in *uucp*(1). The argument *user* is the login name of someone on the specified system.

Two options are available:

—p Copy the source file into the spool directory before transmission.

—m Send mail to the sender when the copy is complete.

The files (or subtrees if directories are specified) are sent to a public directory (PUBDIR) on *system*. Specifically, the files are sent to the directory

PUBDIR/receive/user/fssystem,

where *user* is the recipient, and *fssystem* is the sending system.

The recipient is notified by *mail* of the arrival of files.

## uupick

The command *uupick* may be used by a user to accept or reject the files transmitted to the user. Specifically, *uupick* searches PUBDIR on the user's system for files sent to the user. For each entry (file or directory) found, one of the following messages is printed on the standard output:

from system: dir dirname ?

from system: file file-name ?

The command *uupick* then reads a line from the standard input to determine the disposition of the file. The user's possible responses are:

<newline>

Go on to next entry.

d Delete the entry.

m [ *dir* ]

Move the entry to named directory *dir*. If *dir* is not specified as a complete path name a destination relative to the current directory is assumed. If no destination is given, the default is the current directory.

**a** [ *dir* ]

Same as *m* except moving all the files sent from *system*.

**p** Print the content of the file to standard output.

**q** Stop and exit.

**<EOF>**

Same as *q*.

**!command**

Escape to the command interpreter to execute *command*.

**\*** Print a usage summary for *uuto*.

The command *uupick* invoked with the *—s system* option will only search for files (and list any found) sent from *system*.

## SEE ALSO

*mail*(1), *uucp*(1), *uustat*(1), *uux*(1).

## APPLICATION USAGE

Typical implementations of this utility require a communications line configured to use the *termio*(7) interface. On systems where none of these lines are available, this utility may not be present.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.



## NAME

uux — remote command execution

## SYNOPSIS

**uux [ options ] command-string**

UN

## DESCRIPTION

The command *uux* will gather zero or more files from various systems, execute a command on a specified system, and then send the standard output of the command to a file on a specified system.

The *command-string* is made up of one or more arguments that are similar to normal command arguments, except that the command and any file names may be prefixed by *system-name!*. A null *system-name* is interpreted as the local system.

The following statements are relevant if *sh*(1) is the command interpreter.

The metacharacter *\** will not give the desired result.

The redirection tokens *>>* and *<<* are not implemented.

A file name may be specified as for *uucp*; it may be a full path name, a path name preceded by *~name* (which is replaced by the corresponding login directory), a path name specified as *~/dest* (*dest* is prefixed by *PUBDIR*), or a simple file name (which is prefixed by the current directory). See *uucp*(1) for the details.

As an example, the command

```
uux "!diff usg! /usr/dan/file1 pwba! /a4/dan/file2 > !~/dan/file.diff"
```

will get the *file1* and *file2* files from the *usg* and *pwba* machines, execute *diff*, and put the results in *file.diff* in the local *PUBDIR/dan* directory. (*PUBDIR* is the *uucp* public directory on the local system.)

The execution of commands on remote systems takes place in an execution directory known to the *uucp* system. All files required for the execution will be put into this directory unless they already reside on that machine. Therefore, the non-local file names (without path or machine reference) must be unique within the *uux* request. The following command will not work:

```
uux "a!diff b!/usr/dan/xyz c!/usr/dan/xyz > !xyz.diff"
```

because the file *xyz* will be copied from the *b* system as well as the *c* system, causing a name conflict. The command:

```
uux "a!diff a!/usr/dan/xyz c!/usr/dan/xyz > !xyz.diff"
```

will work (provided *diff* is a permitted command), because the local file *xyz* (which is not copied) does not conflict with the copied file *xyz* from the *c* system.

Any characters special to the command interpreter should be quoted either by quoting the entire *command-string* or quoting the special characters as individual arguments.



The command *uux* will attempt to get all files to the execution system. For files that are output files, the file name must be escaped using parentheses. For example, the command:

```
uux alcut —f1 b!/usr/file \(c!/usr/file\)
```

gets */usr/file* from system *b*, sends it to system *a*, performs a *cut* command on that file, and sends the result of the *cut* command to system *c*.

The command *uux* will notify the user (by mail) if the requested command on the remote system was disallowed. This notification can be turned off by the *—n* option. The response comes by mail from the remote machine.

The following *options* are interpreted by *uux*:

— The standard input to *uux* is made the standard input to the *command-string*.

UN —j Output the job identification ASCII string on the standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.

—n Do not notify the user if the command fails.

#### SEE ALSO

*uucp*(1), *uustat*(1).

#### APPLICATION USAGE

Note that, for security reasons, many installations will limit the list of commands executable on behalf of an incoming request from *uux*. Many sites will permit little more than the receipt of mail via *uux*.

Only the first command of a pipeline, see *sh*(1), may have a *system-name!*. All other commands are executed on the system of the first command.

Typical implementations of this utility require a communications line configured to use the *termio*(7) interface. On systems where none of these lines are available, this utility may not be present.

#### CHANGE HISTORY

First released in Issue 2.

#### Issue 2

Derived from the entry in Issue 2 of the SVID.

## NAME

val — validate SCCS file

## SYNOPSIS

val —

val [—s] [—rSID] [—mname] [—ytype] file...

## DESCRIPTION

The command *val* determines if the specified *file* is an SCCS file meeting the characteristics specified by the options. The arguments may appear in any order.

*Val* has a special argument, —, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

*Val* generates diagnostic messages on the standard output for each command line and file processed, and also returns a single 8-bit code upon exit as described below.

The options are defined as follows. The effects of any option apply independently to each named file on the command line.

—s Silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

—rSID *SID* (SCCS Identification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e.g., —r1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (e.g., —r1.0 or —r1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

—mname

*name* is compared with the SCCS %M% keyword in *file*.

—ytype

*type* is compared with the SCCS %Y% keyword in *file*.

## EXIT STATUS

The 8-bit code returned by *val* is a disjunction of the possible errors, i.e., it can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

- |       |   |                                     |
|-------|---|-------------------------------------|
| bit 0 | = | missing file argument;              |
| bit 1 | = | unknown or duplicate option;        |
| bit 2 | = | corrupted SCCS file;                |
| bit 3 | = | cannot open file or file not SCCS;  |
| bit 4 | = | <i>SID</i> is invalid or ambiguous; |
| bit 5 | = | <i>SID</i> does not exist;          |
| bit 6 | = | %Y%, —y mismatch;                   |
| bit 7 | = | %M%, —m mismatch;                   |



## VAL(1D)

Utilities

Note that *val* can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned: a logical *or* of the codes generated for each command line and file processed.

### SEE ALSO

admin(1D), delta(1D), get(1D), prs(1D).

### CHANGE HISTORY

First released in Issue 2.

### Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The `—r` option in the SYNOPSIS has been corrected from "SCCS" to "SID".



## NAME

*vi* — screen-oriented (visual) display editor

## SYNOPSIS

*vi* [*-r*] [*-l*] [*-wn*] [*-R*] [*+command*] [*file*]

OP

## DESCRIPTION

*Vi* (visual) is a display-oriented text editor. It is based on the underlying line editor *ex*(1): it is possible to switch back and forth between the two, and to execute *ex* commands from within *vi*.

When using *vi*, the terminal screen acts as window into the file being edited. Changes made to the file are reflected in the screen display; the position of the cursor on the screen indicates the position within the file.

The environment variable *TERM* must give the terminal type; the terminal must be defined in a terminal description database. As for *ex*, editor initialisation scripts can be placed in the environment variable *EXINIT*, or the file *.exrc* in the current or home directory.

## Options

The following options are interpreted by *vi*:

- r* Recover *file* after an editor or system crash. If *file* is not specified a list of all saved files will be printed.
- l* set *lisp* mode (see **Edit Options** in *ex*(1)).
- wn* Set the default window size to *n*.
- R* Read only mode; the *readonly* flag is set, preventing accidental overwriting of the file.
- +command*

The specified *ex command* is interpreted before editing begins.

## Vi Commands

## General Remarks

See *ex*(1) for the complete description of *ex*. Only the *visual* mode of the editor is described here.

At the beginning, *vi* is in the *command* mode; the *input* mode is entered by several commands used to insert or change text. In input mode, ESC (escape) is used to leave input mode; in command mode, it is used to cancel a partial command; the terminal bell is sounded if the editor is not in input mode and there is no partially entered command.

The last (bottom) line of the screen is used to echo the input for search commands (*/* and *?*), for *ex* commands (*:*), and system commands (*!*). It is also used to report errors or print other messages.

Receipt of SIGINT during text input, or during the input of a command on the bottom line, terminates the input (or cancels the command) and returns the editor to command mode. During command mode, SIGINT causes the bell to be sounded; in general the bell indicates an error (such as unrecognised key).

Lines displayed on the screen containing only a ~ indicate that the last line above them is the last line of the file (the ~ lines are past the end of the file). On a terminal with limited local intelligence, there may be lines on the screen marked with an @: these indicate space on the screen not corresponding to lines in the file. (These lines may be removed by entering a ^R, forcing the editor to retype the screen without these holes.)

#### Command Summary

Most commands accept a preceding number as an argument, either to give a size or position (for display or movement commands), or as a repeat count (for commands that change text). For simplicity, this optional argument will be referred to as *count* when its effect is described.

The following operators can be followed by a movement command, in order to specify an extent of text to be affected: *c*, *d*, *y*, *<*, *>*, *!*, and *=*. The region specified is from the current cursor position to just before the cursor position indicated by the move. If the command operates on lines only, then all the lines which fall partly or wholly within this region are affected. Otherwise the exact marked region is affected.

In the following, control characters are indicated in the form “^X”, which stands for “control-X”.

Unless otherwise specified, the commands are interpreted in command mode and have no special effect in input mode.

- ^B    Scrolls backward to display the window above the current one. A count specifies the number of windows to go back. Two lines of overlap are kept if possible.
- ^D    Scrolls forward a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future ^D and ^U commands.  
  
      In input mode, backs *shiftwidth* spaces over the indentation provided by *autoindent* or ^T.
- ^E    Scrolls forward one line, leaving the cursor where it is if possible.
- ^F    Scrolls forward to display the window below the current one. A count specifies the number of windows to go forward. Two lines of overlap are kept if possible.
- ^G    Prints the current file name and other information, including the number of lines and the current position. (Equivalent to the *ex* command *f*.)



- ^H** Moves one space to the left (stops at the left margin). A count specifies the number of spaces to back up. (Same as *h*.)
- In input mode, backs over the last input character without erasing it.
- ^J** Moves the cursor down one line in the same column. A count specifies the number of lines to move down. (Same as **^N** and *j*.)
- ^L** Clears and redraws the screen. (Used when the screen is scrambled for any reason.)
- ^M** Moves to the first non-white character in the next line. A count specifies the number of lines to go forward.
- ^N** Same as **^J** and *j*.
- ^P** Moves the cursor up one line in the same column. A count specifies the number of lines to move up. (Same as *k*.)
- ^R** Redraws the current screen, eliminating the false lines marked with @ (which do not correspond to actual lines in the file).
- ^T** In input mode, if at the beginning of the line or preceded only by white space, inserts *shiftwidth* white space. This inserted space can only be backed over using **^D**.
- ^U** Scrolls up a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands.
- ^V** In input mode, quotes the next character to make it possible to insert special characters (including ESC) into the file.
- ^W** In input mode, backs up one word; the deleted characters remain on the display.
- ^Y** Scrolls backward one line, leaving the cursor where it is if possible.
- ^[\_** Cancels a partially formed command; sounds the bell if there is none.
- In input mode, terminates input mode.
- When entering a command on the bottom line of the screen (ex command line or search pattern with \ or ?), terminates input and executes command.

### Space

- Moves one space to the right (stops at the end of the line). A count specifies the number of spaces to go forward. (Same as *l*.)
- !** An operator which passes specified lines from the buffer as standard input to the specified system command, and replaces those lines with the standard output from the command. The **!** is followed by a movement command specifying the lines to be passed (lines from the current position to the end of the movement) and then the command (terminated as usual by a return). A count preceding the **!** is passed on to the movement command after **!**.



Doubling `!` and preceding it by a count causes that many lines, starting with the current line, to be passed.

- `"` Precedes a named buffer specification. There are named buffers 1-9 in which the editor places deleted text. The named buffers `a-z` are available to the user for saving deleted or yanked text, see also `y`, below.
- `$` Moves to the end of the current line. A count specifies the number of lines to go forward (e.g., `2$` goes to the end of the next line).
- `%` Moves to the parenthesis or brace which matches the parenthesis or brace at the current cursor position.
- `&` Same as the `ex` command `&` (repeats previous substitute command).

When followed by a ```, returns to the previous context, placing the cursor at the beginning of the line. (The previous context is set whenever a non-relative move is made.) When followed by a letter `a-z`, returns to the line marked with that letter (see the `m` command), at the first non-white character in the line.

When used with an operator such as `d` to specify an extent of text, the operation takes place over complete lines. (See also ```.)

When followed by a ```, returns to the previous context, placing the cursor at the character position marked. (The previous context is set whenever a non-relative move is made.) When followed by a letter `a-z`, returns to the line marked with that letter (see the `m` command), at the character position marked.

When used with an operator such as `d` to specify an extent of text, the operation takes place from the exact marked place to the current position within the line. (See also ```.)

`[[` Backs up to the previous section boundary. A section is defined by the value of the `sections` option. Lines which start with a formfeed (`^L` character) or `{` also stop `[[`.

If the option `lisp` is set, stops at each `(` at the beginning of a line.

`]]` Moves forward to a section boundary (see `[[`).

`^` Moves to the first non-white position on the current line.

`(` Moves backward to the beginning of a sentence. A sentence ends at a `. !` or `?` which is followed by either the end of a line or by two spaces. Any number of closing `) ] "` and ``` characters may appear between the `. !` or `?` and the spaces or end of line. A count moves back that many sentences.

If the `lisp` option is set, moves to the beginning of a `lisp` s-expression. Sentences also begin at paragraph and section boundaries (see `{` and `[[`).

- ) Moves forward to the beginning of a sentence. A count moves forward that many sentences. (See (.)
- { Moves back to the beginning of the preceding paragraph. A paragraph is defined by the value of the *paragraphs* option. A completely empty line, and a section boundary (see [[ above), are also taken to begin paragraphs. A count specifies the number of paragraphs to move backward.
- } Moves forward to the beginning of the next paragraph. A count specifies the number of paragraphs to move forward. (See { .)
- | Requires a count; the cursor is placed in that column (if possible).
- + Moves to the first non-white character in the next line. A count specifies the number of lines to go forward. (Same as ^M.)
- , Reverse of the last *f F t* or *T* command, looking the other way in the current line. A count is equivalent to repeating the search that many times.
- Moves to the first non-white character in the previous line. A count specifies how many lines to move back.
- . Repeats the last command which changed the buffer. A count is passed on to the command being repeated.
- / Reads a string from the last line on the screen, interprets it as a regular expression, and scans forward for the next occurrence of a matching string. The search begins when return is entered to terminate the pattern; it may be terminated by sending SIGINT.  
  
When used with an operator to specify an extent of text, the defined region is from the current cursor position to the beginning of the matched string. Whole lines may be specified by giving an offset from the matched line (using a closing / followed by a *+n* or *-n*).
- 0 Moves to the first character on the current line. (Is not interpreted as a command when preceded by a non-zero digit.)
- : Begins an *ex* command. The :, as well as the entered command, is echoed on the bottom line; it is executed when the input is terminated by entering a return.
- ; Repeats the last single character find using *f F t* or *T*. A count is equivalent to repeating the search that many times.
- < An operator which shifts lines left one *shiftwidth*. May be followed by a move to specify lines. A count is passed through to the move command.  
  
When repeated ( << ), shifts the current line (or count lines starting at the current one).
- > An operator which shifts lines right one *shiftwidth*. (See <.)



- = If the *lisp* option is set, then reindents the specified lines, as though they were typed in with *lisp* and *autoindent* set. May be preceded by a count to indicate how many lines to process, or followed by a move command for the same purpose.
- ? Scans backwards, the reverse of / . (See / .)
- A Appends at the end of line. (Same as \$a.)
- B Backs up a word, where a word is any non-blank sequence, placing the cursor at the beginning of the word. A count gives the number of words to go back.
- C Changes the rest of the text on the current line. (Same as c\$.)
- D Deletes the rest of the text on the current line. (Same as d\$.)
- E Moves forward to the end of a word, where a word is any non-blank sequence. A count gives the number of words to go forward.
- F Must be followed by a single character; scans backwards in the current line for that character, moving the cursor to it if found. A count is equivalent to repeating the search that many times.
- G Goes to the line number given as preceding argument, or the end of the file if no preceding count is given.
- H Moves the cursor to the top line on the screen. If a count is given, then the cursor is moved to that line on the screen, counting from the top. The cursor is placed on the first non-white character on the line. If used as the target of an operator, full lines are affected.
- I Inserts at the beginning of a line. (Same as ^ followed by i.)
- J Joins the current line with the next one, supplying appropriate white space: one space between words, two spaces after a period, and no spaces at all if the first character of the next line is ). A count causes that many lines to be joined rather than two.
- L Moves the cursor to the first non-white character of the last line on the screen. A count moves to that line counting from the bottom. When used with an operator, whole lines are affected.
- M Moves the cursor to the middle line on the screen, at the first non-white position on the line.
- N Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of n.
- O Opens a new line above the current line and enters input mode.



- P** Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted just before the cursor.
- May be preceded by a named buffer specification ("*x*"), to retrieve the contents of the buffer.
- Q** Quits from *vi* and enters *ex* command mode.
- R** Replaces characters on the screen with characters entered, until the input is terminated with ESC.
- S** Changes whole lines (same as *cc*). A count changes that many lines.
- T** Must be followed by a single character; scans backwards in the current line for that character, and if found, places the cursor just after that character. A count is equivalent to repeating the search that many times.
- U** Restores the current line to its state before the cursor was last moved to it.
- W** Moves forward to the beginning of a word in the current line, where a word is a sequence of non-blank characters. A count specifies the number of words to move forward.
- X** Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.
- Y** Places (yanks) a copy of the current line into the unnamed buffer (same as *yy*). A count copies that many lines. May be preceded by a buffer name to put the copied line(s) in that buffer.
- ZZ** Exits the editor, writing out the buffer if it was changed since the last write. (Same as the *ex* command *x*.)
- a** Enters input mode, appending the entered text after the current cursor position. A count causes the inserted text to be replicated that many times, but only if the inserted text is all on one line.
- b** Backs up to the previous beginning of a word in the current line. A word is a sequence of alphanumerics or a sequence of special characters. A count repeats the effect.
- c** Must be followed by a movement command. Deletes the specified region of text, and enters input mode to replace it with the entered text. If more than part of a single line is affected, the deleted text is saved in the numeric buffers. If only part of the current line is affected, then the last character to be deleted is marked with a *\$*. A count is passed through to the move command. If the command is *cc*, the whole of the current line is changed.

- d Must be followed by a movement command. Deletes the specified region of text. If more than part of a line is affected, the text is saved in the numeric buffers. A count is passed through to the move command. If the command is *dd*, the whole of the current line is deleted.
- e Moves forward to the next word-end, defined as for *b*. A count repeats the effect.
- f Must be followed by a single character; scans the rest of the current line for that character, and moves the cursor to it if found. A count repeats the find that many times.
- h Moves the cursor one character to the left. (Same as *^H*.) A count repeats the effect.
- i Enters input mode, inserting the entered text before the cursor. (See *a*.)
- j Moves the cursor one line down in the same column. (Same as *^J* and *^N*.)
- k Moves the cursor one line up. (Same as *^P*.)
- l Moves the cursor one character to the right. (Same as *<space>*.)
- m Must be followed by a single lower case letter *x*; marks the current position of the cursor with that letter. The exact position is referred to by *x*; the line is referred to by *^x*.
- n Repeats the last / or ? scanning commands.
- o Opens a line below the current line and enters input mode; otherwise like *O*.
- p Puts text after/below the cursor; otherwise like *P*.
- r Must be followed by a single character; the character under the cursor is replaced by the specified one. (The new character may be a newline.) A count replaces each of the following count characters with the single character given.
- s Deletes the single character under the cursor and enters input mode; the entered text replaces the deleted character. A count specifies how many characters from the current line are changed. The last character to be changed is marked with a *\$*, as for *c*.
- t Must be followed by a single character; scans the rest of the line for that character. The cursor is moved to just before the character, if it is found. A count is equivalent to repeating the search that many times.
- u Reverses the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers.



- w Moves forward to the beginning of the next word, where word is the same as in *b*. A count specifies how many words to go forward.
- x Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.
- y Must be followed by a movement command; the specified text is copied (yanked) into the unnamed temporary buffer. If preceded by a named buffer specification, "x, the text is placed in that buffer also. If the command is yy, the whole of the current line is deleted.
- z Redraws the screen with the current line placed as specified by the following character: <return> specifies the top of the screen, . the centre of the screen, and — the bottom of the screen. A count may be given after the z and before the following character to specify the new screen size for the redraw. A count before the z gives the number of the line to place in the centre of the screen instead of the default current line.

## SEE ALSO

ex(1).

## APPLICATION USAGE

As this editor depends on the *optional curses* package, applications should be aware that it may not be present, or not function with certain classes of terminal.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

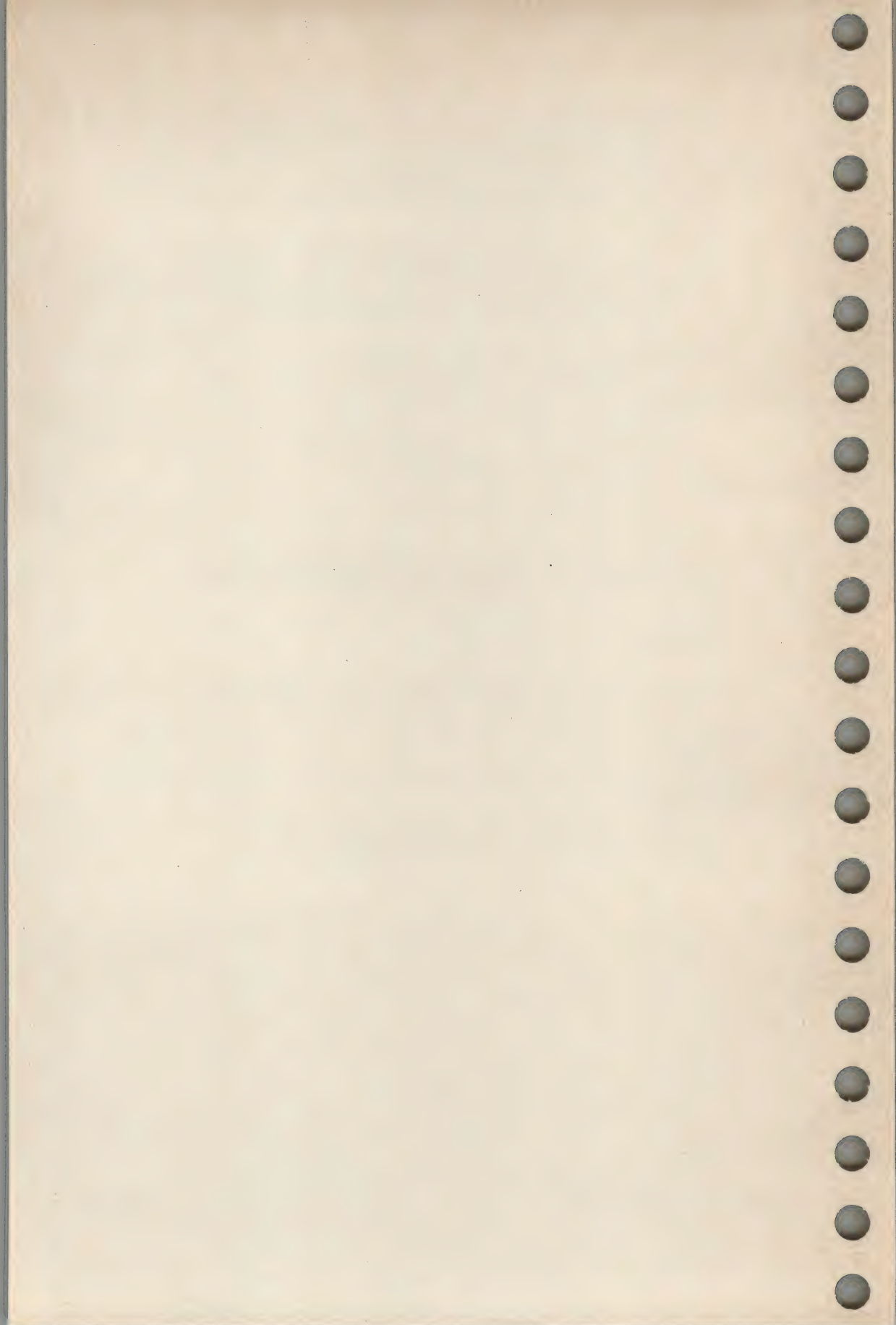
Derived from the entry in Issue 2 of the SVID with the following changes:

The sentence "Must be followed by a movement command." has been added to the description of the *c* and *d* commands.

In the description of the *e* command, the words "to the end of the next word" have been changed to "to the next word-end".

Descriptions of the *cc*, *dd* and *yy* commands have been added.





## NAME

wait — await completion of process

## SYNOPSIS

wait [ pid ]

## DESCRIPTION

With no argument, *wait* waits until all processes started with *&* by the current shell have completed, and reports on abnormal terminations. If a numeric argument *pid* is given, and is the process ID of a background process, then *wait* waits until that process has completed. Otherwise, if *pid* is not a background process, *wait* waits until all background processes have completed.

## SEE ALSO

sh(1), wait(2).

## APPLICATION USAGE

This is always a shell built in command.

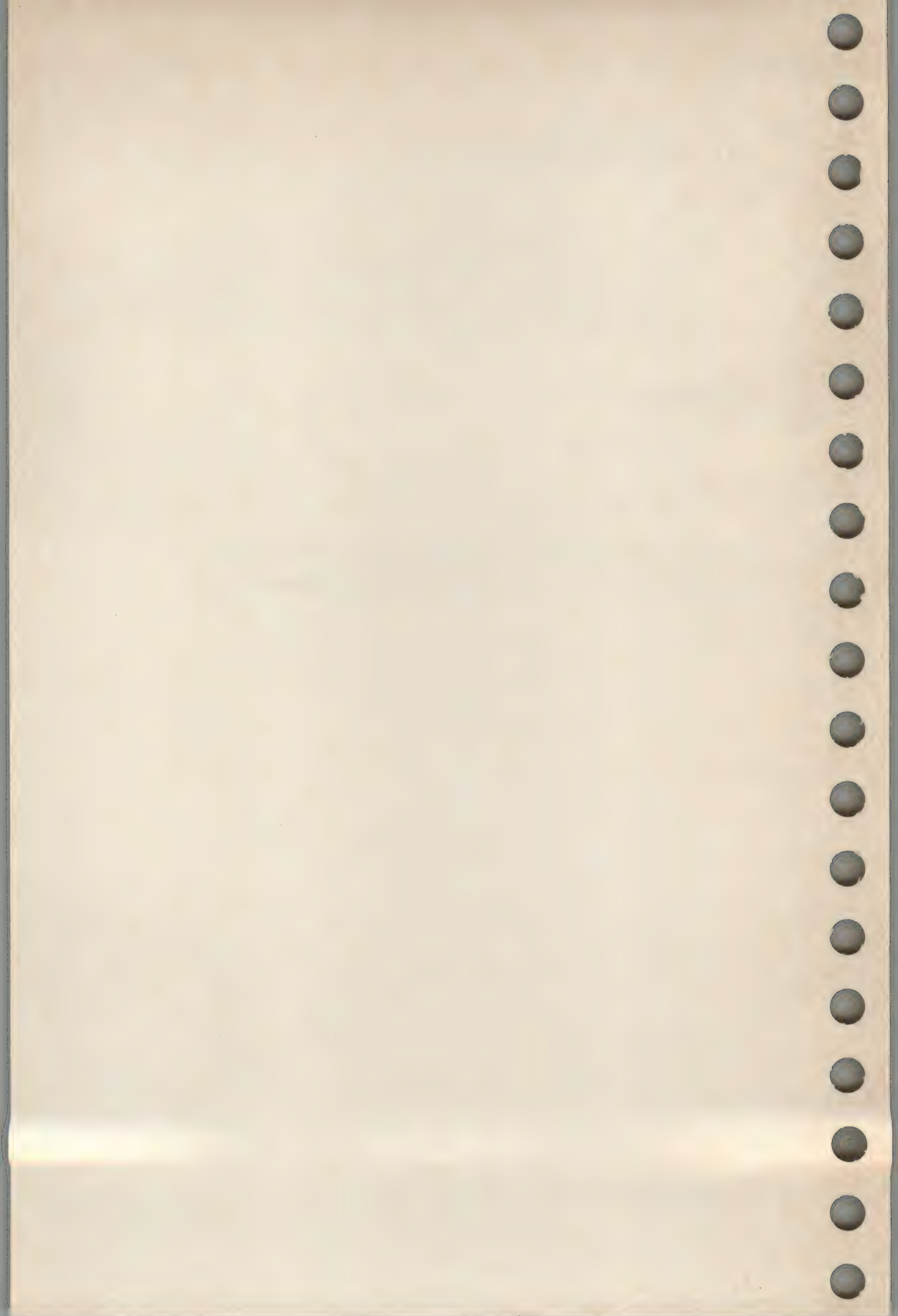
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The words "by the current shell" have been added to the description.





## NAME

wall — write to all users

## SYNOPSIS

wall

## DESCRIPTION

The command *wall* reads a message on the standard input until an end-of-file. It then prints this message on the terminals of all users currently logged-in, preceded by an announcement indicating who sent the message.

The sender must be super-user to override any protections the users may have invoked, see *mesg*(1).

## SEE ALSO

*mesg*(1), *write*(1).

## APPLICATION USAGE

The command *wall* is used to warn all users, typically prior to shutting down the system.

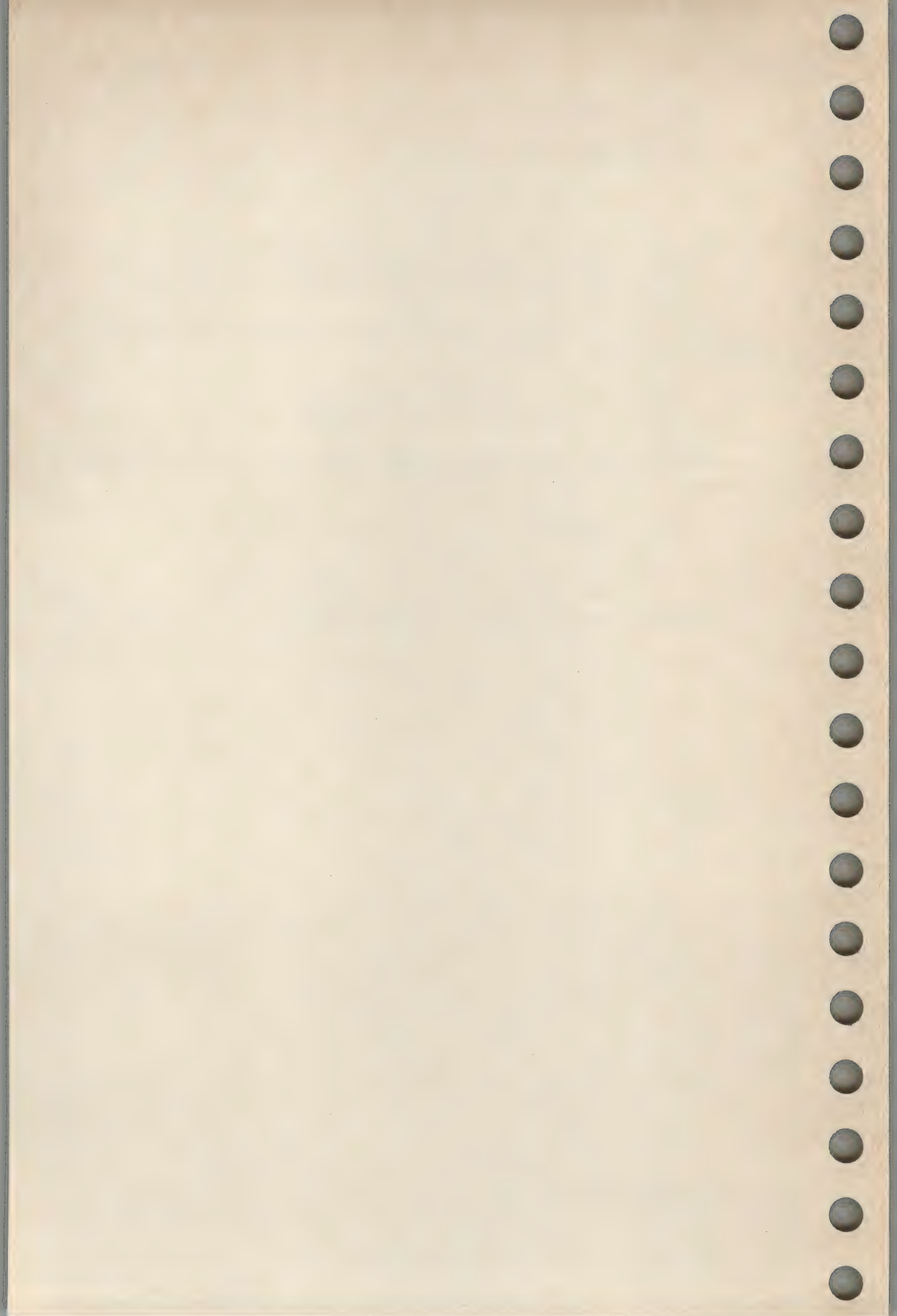
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The pathname */etc* has been removed from the SYNOPSIS.





## NAME

wc — word count

## SYNOPSIS

**wc [—lwc] [file ...]**

OF

## DESCRIPTION

The command *wc* counts lines, words and characters in the *file* or files named, or in the standard input if no *file* appears. It also keeps a total count for all named files. A word is defined as a maximal string of characters delimited by spaces, tabs, or newlines.

The options *l*, *w*, and *c* may be used in any combination to specify that a subset of lines, words, and characters are to be reported. The default is *—lwc*.

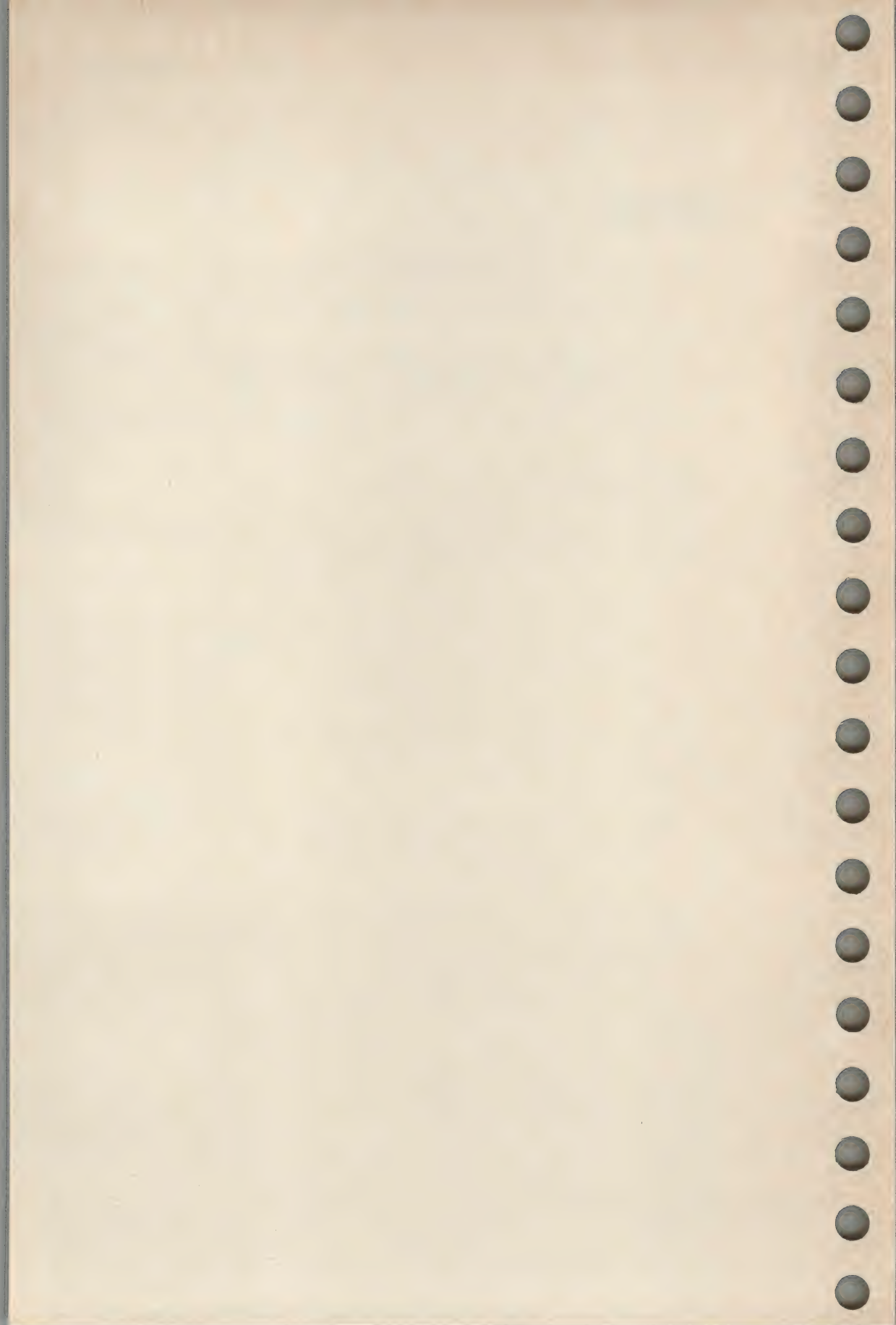
When *file* is specified on the command line, their names will be printed along with the counts.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

what — identify SCCS files

## SYNOPSIS

what [—s] file...

## DESCRIPTION

The command *what* searches the given files for all occurrences of the pattern that *get*, see *get*(1D), substitutes for %Z% (@(#)) and prints out what follows until the first " , > , newline, \ , or NUL character.

*What* is intended to be used in conjunction with the SCCS command *get*, which automatically inserts identifying information, but it can also be used where the information is inserted manually.

There is only one option:

—s   Quit after finding the first occurrence of pattern in each file.

## EXAMPLES

If the C program in file *f.c* contains

```
char ident[] = "@(#)identification information";
```

and *f.c* is compiled to yield *f.o* and *a.out*, then the command

```
what f.c f.o a.out
```

will print

```
f.c:      identification information
```

```
...
```

```
f.o:      identification information
```

```
...
```

```
a.out:    identification information
```

```
...
```

If only one *file* argument is given, the *filename:* information is not output.

## EXIT STATUS

Exit status is:

0   any matches were found

1   otherwise

## SEE ALSO

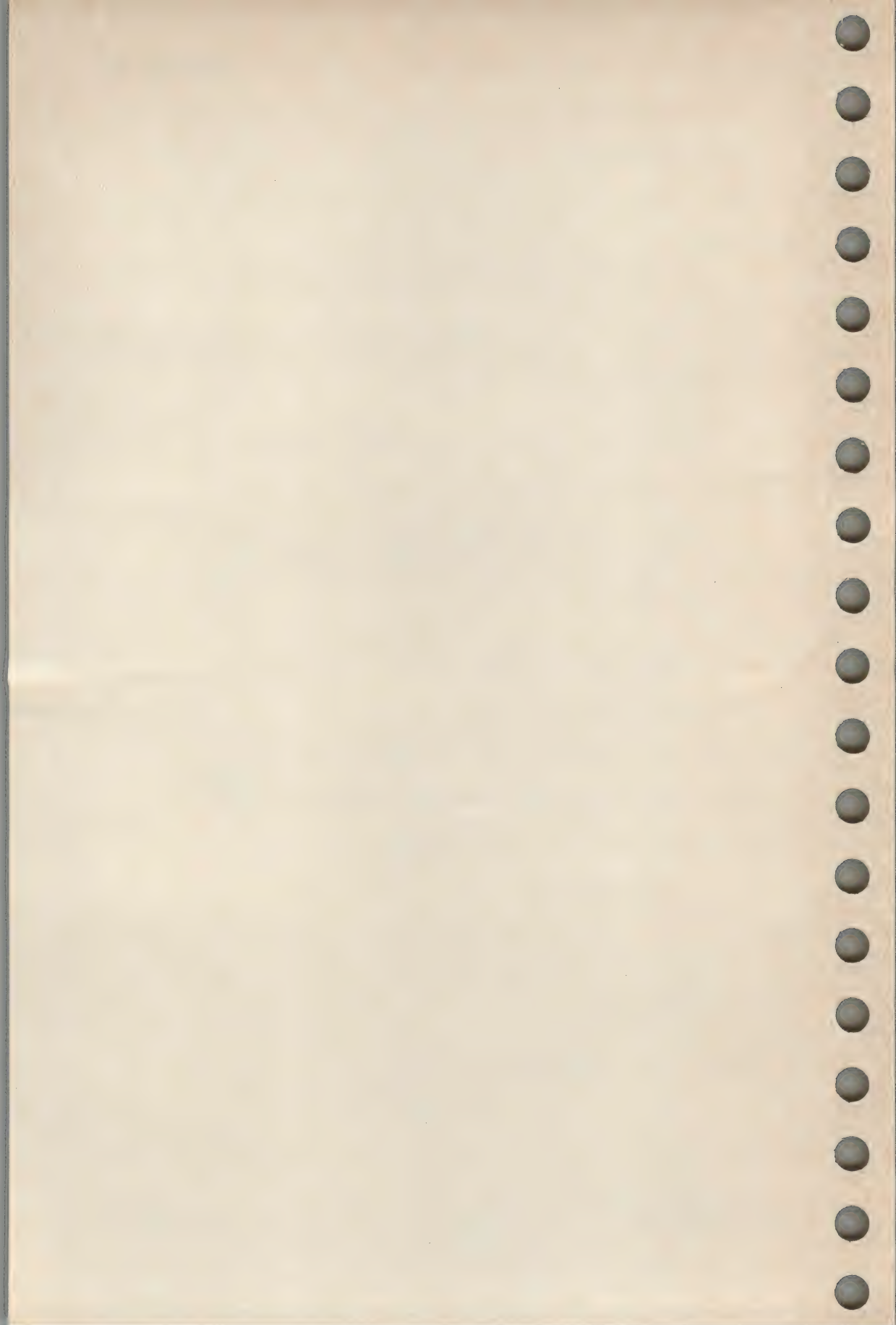
*get*(1D).

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.





## NAME

who — who is on the system

## SYNOPSIS

```
who
who [ options ] [ file ]
who am i
who am I
```

OF  
OF UN  
OF  
OF

## DESCRIPTION

The command *who* can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process ID of the command interpreter for each current system user. It examines the */etc/utmp* file to obtain its information. If *file* is given, that file is examined instead.

The command *who* with the *am i* or *am I* option identifies the invoking user.

Except for the default —s option, the general format for output entries is:

```
name [state] line time activity pid [comment] [exit]
```

With *options*, *who* can list logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the *init* process. These *options* are:

- u This option lists only those users who are currently logged in. The *name* is the user's login name. The *line* is the name of the line as found in the directory */dev*. The *time* is the time that the user logged in. The *activity* is the number of hours and minutes since activity last occurred on that particular line. A dot (.) indicates that the terminal has seen activity in the last minute and is therefore "current". If more than twenty-four hours have elapsed or the line has not been used since boot time, the entry is marked *old*. This field is useful when trying to determine whether a person is working at the terminal or not. The *pid* is the process ID of the user's login process.
- T This option is the same as the —u option, except that the *state* of the terminal line is printed. The *state* describes whether someone else can write to that terminal. A + appears if the terminal is writable by anyone; a - appears if it is not. The super-user can write to all lines having a + or a - in the *state* field. If a bad line is encountered, a ? is printed. See *mesg(1)*.
- l This option lists only those lines on which the system is waiting for someone to login. The *name* field is LOGIN in such cases. Other fields are the same as for user entries except that the *state* field does not exist.
- H This option will print column headings above the regular output.
- q This is a quick *who*, displaying only the names and the number of users currently logged on. When this option is used, all other options are ignored.
- p This option lists any other process which is currently active and has been previously spawned by *init*.

- d This option displays all processes that have expired and not been respawned by *init*. The *exit* field appears for dead processes and contains the termination and exit values of the dead process. This can be useful in determining why a process terminated.
- b This option indicates the time and date of the last reboot.
- r This option indicates the current *run-level* of the *init* process.
- t This option indicates the last change to the system clock.
- a This option processes */etc/utmp* or the named *file* with all options turned on.
- s This option is the default and lists only the *name*, *line*, and *time* fields.

## FILES

*/etc/utmp* default source of information for *who*.

## SEE ALSO

*mesg(1)*.

## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID.



## NAME

write — write to another user

## SYNOPSIS

write user [ terminal ]

## DESCRIPTION

The command *write* copies its standard input to the terminal of another user. When first called, *write* sends a message to the user addressed.

The recipient of the message should write back, by typing *write sender-login-id*, on receipt of the initial message. Whatever each user types (except for command escapes, see below) is printed on the other user's terminal, until an end-of-file or an interrupt is sent. At that point *write* writes

EOT

on the other terminal and exits. The recipient can also stop further messages from coming in by executing *mesg n*.

To write to a user who is logged in more than once, the *terminal* argument may be used to indicate which terminal to send to (e.g., *tty00*); otherwise, the first writable instance of the user found in */etc/utmp* is assumed and an informational message is written.

A user may deny or grant write permission by use of the *mesg* command. Certain commands disallow messages in order to prevent interference with their output. However, if the sender has super-user permissions, messages can be forced onto a write-inhibited terminal.

If the character *!* is found at the beginning of a line, *write* calls the command interpreter to execute the rest of the line as a command.

## FILES

*/etc/utmp*          history file of user logins.

## ERRORS

The following errors are reported:

the user addressed is not logged on.

the user addressed denies write permission, see *mesg(1)*.

the user's terminal is set to *mesg n* and the recipient cannot respond.

the recipient changes permission (*mesg n*) after *write* had begun.

## SEE ALSO

*mesg(1)*, *who(1)*.

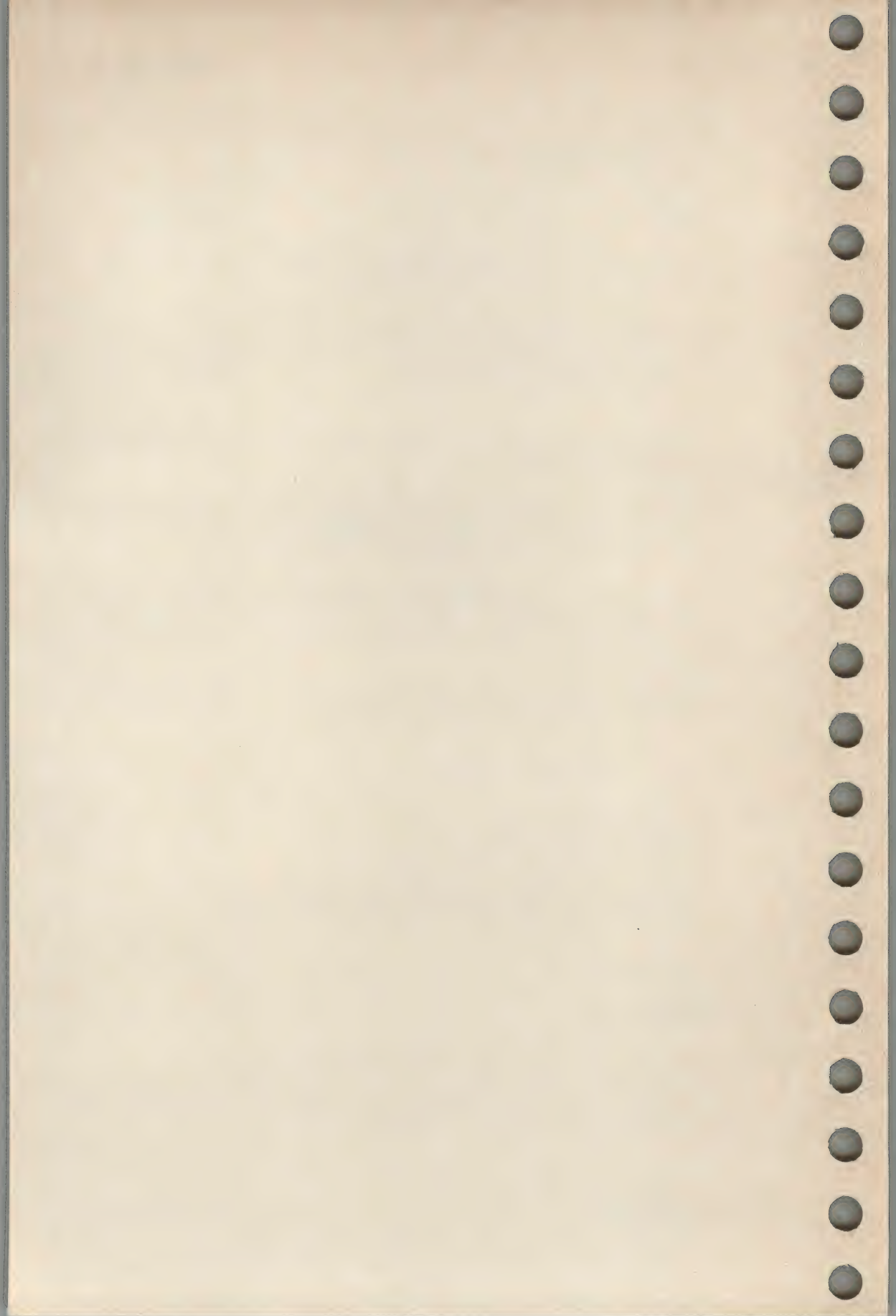
## CHANGE HISTORY

First released in Issue 2.

## Issue 2

Derived from the entry in Issue 2 of the SVID with the following change:

The first sentence of the **DESCRIPTION** has been changed from: "The command *write* copies lines from the user's terminal to that of another user."





## NAME

xargs — construct argument list(s) and execute command

## SYNOPSIS

xargs [ option ... ] [ command [ initial-argument ... ] ]

## DESCRIPTION

*Xargs* combines the fixed *initial-argument* or arguments with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the *options* specified.

If *command* is omitted, *echo* is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or newlines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings a backslash quotes the next character.

Each argument list is constructed starting with the *initial-arguments*, followed by some number of arguments read from standard input (Exception: see *—i*). Options *—i*, *—l*, and *—n* determine how arguments are selected for each command invocation. When none of these options are coded, the *initial-arguments* are followed by arguments read continuously from standard input until an internal buffer is full, and then *command* is executed with the accumulated args. This process is repeated until there are no more args. When there are conflicts (e.g., *—l* vs. *—n*), the last option has precedence. The recognised *options* are:

*—l*number

*Command* is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first newline unless the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If *number* is omitted, 1 is assumed. Option *—x* is forced.

*—i*replstr

Insert mode: *command* is executed for each line from standard input, taking the entire line as a single arg, inserting it in *initial-arguments* for each occurrence of *replstr*. A maximum of 5 arguments in *initial-arguments* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters, and option *—x* is also forced. { } is assumed for *replstr* if not specified.

**—nnumber**

Execute *command* using as many standard input arguments as possible, up to *number* arguments maximum. Fewer arguments will be used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* arguments remaining. If option **—x** is also invoked, each *number* arguments must fit in the *size* limitation, otherwise *xargs* terminates execution.

**—t** Trace mode: The *command* and each constructed argument list are echoed to standard error just prior to their execution.**—p** Prompt mode: The user is asked whether to execute *command* at each invocation. Trace mode (**—t**) is turned on to print the command instance to be executed, followed by a *?... prompt*. A reply of *y* (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of *command*.**—x** Causes *xargs* to terminate if any argument list would be greater than *size* characters; **—x** is forced by the options **—i** and **—l**. When neither of the options **—i**, **—l**, or **—n** are coded, the total length of all arguments must be within the *size* limit.**—ssize**

The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If **—s** is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

**—eofstr**

*eofstr* is taken as the logical end-of-file string. Underscore ( *\_* ) is assumed for the logical EOF string if **—e** is not invoked. The option **—e** with no *eofstr* coded turns off the logical EOF string capability (underscore is taken literally). *Xargs* reads standard input until either end-of-file or the logical EOF string is encountered.

*Xargs* will terminate if either it receives a return code of -1 from, or if it cannot execute, *command*. (Thus *command* should explicitly *exit* with an appropriate value to avoid accidentally returning with -1 .)

**EXAMPLES**

1. The following will move all files from directory *\$1* to directory *\$2*, and echo each move command just before doing it:

```
ls $1 | xargs —i —t mv $1/{ } $2/{ }
```

2. The following will combine the output of the parenthesised commands onto one line, which is then echoed to the end of file *log*:

```
(logname; date; echo $0 $*) | xargs >>log
```



3. The user is asked which files in the current directory are to be archived. The files are archived into *arch* (a.) one at a time, or (b.) many at a time.

- a. `ls | xargs —p —l ar r arch`

- b. `ls | xargs —p —l | xargs ar r arch`

4. The following will execute with successive pairs of arguments originally typed as command line arguments:

`echo $* | xargs -n2 diff`

**SEE ALSO**

`echo(1)`.

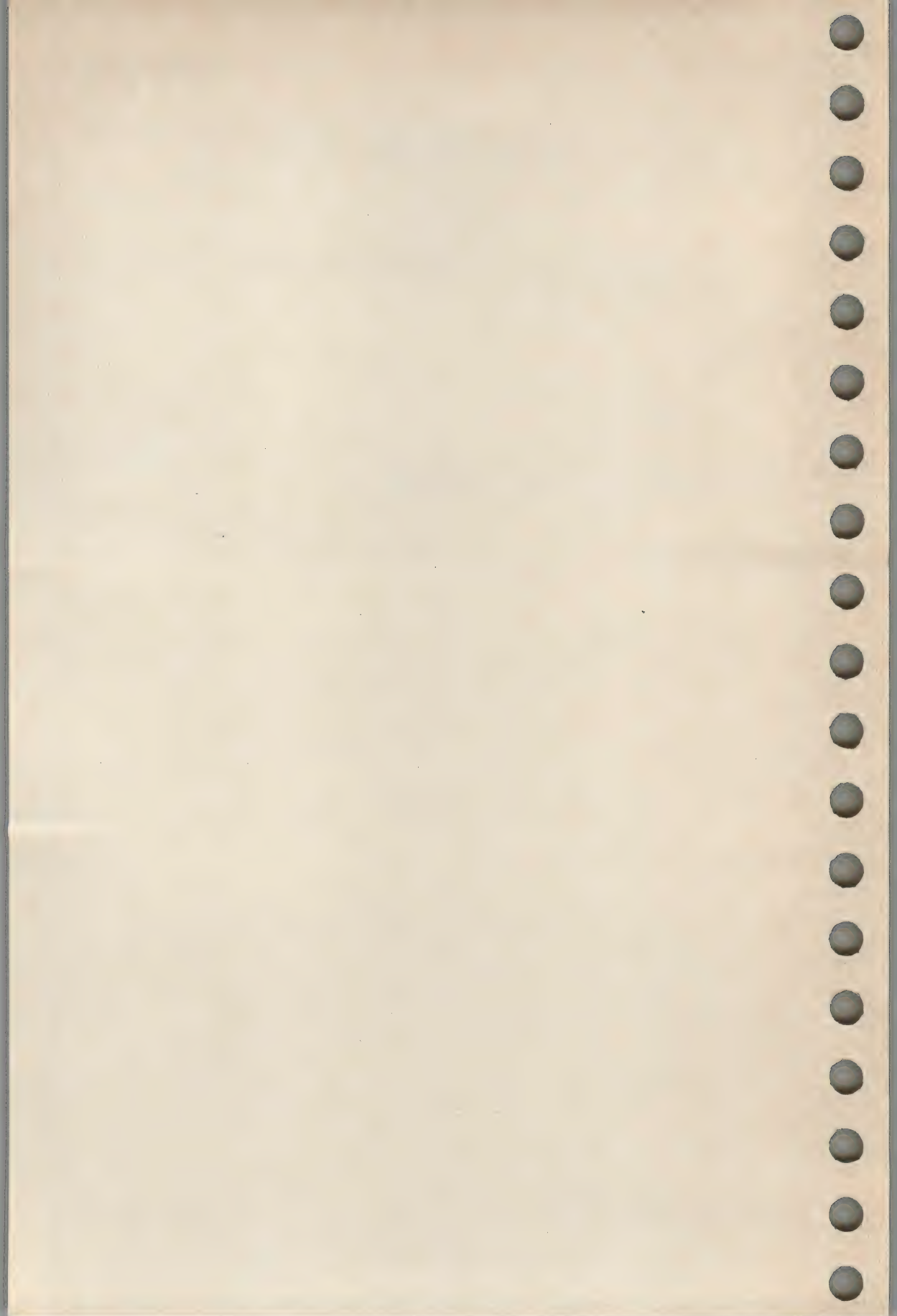
**CHANGE HISTORY**

First released in Issue 2.

**Issue 2**

Derived from the entry in Issue 2 of the SVID.





## NAME

yacc — a compiler-compiler

## SYNOPSIS

yacc [—vd<sup>1</sup>] grammar

## DESCRIPTION

The *yacc* command provides a general tool for describing the input to a program. More precisely, *yacc* converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; built-in precedence rules are used to break ambiguities.

The output file, *y.tab.c*, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyser function, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user (however, see the description of the *yacc* library below); *lex* is useful for creating lexical analysers usable by *yacc*.

If the *—v* option is used, the file *y.output* is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the *—d* option is used, the file *y.tab.h* is generated with the *#define* statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than *y.tab.c* to access the token codes.

MV UN

If the *—t* option is used, the code produced in *y.tab.c* will not contain any *#line* constructs. This should only be used after the grammar and the associated actions are fully debugged.

UN

Runtime debugging code is always generated in *y.tab.c* under conditional compilation control. By default, this code is not included when *y.tab.c* is compiled. However, when *yacc*'s *—t* option is used, this debugging code will be compiled by default. Independent of whether the *—t* option was used, the runtime debugging code is under the control of *YYDEBUG*, a pre-processor symbol. If *YYDEBUG* has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

## Yacc Library

The *yacc* library *liby.a* facilitates the initial use of *yacc* by providing the routines:

```
main()

yyerror(s)
char *s;
```

These routines may be loaded by using the *—ly* option with *cc*. *Main()* just calls *yyparse()*. *Yyerror()* simply prints the string (error message) *s* when a syntax error is detected.



### Yacc Specifications

The *yacc* user constructs a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognised, and a low-level routine to do the basic input. *Yacc* then generates the (integer-valued) function *yyparse*; it in turn calls *yylex*, the lexical analyser, to obtain input tokens.

A structure recognised (and returned) by the lexical analyser is called a *terminal symbol*, here referred to as a *token* (literal characters must also be passed through the lexical analyser, and are also considered tokens). A structure recognised by the parser is called a *nonterminal symbol*. *Name* refers to either tokens or nonterminal symbols.

Every specification file consists of three sections: *declarations*, *grammar rules* and *programs*, separated by double percent marks ("%"). The declarations and programs sections may be empty. If the latter is empty, then the preceding %% mark separating it from the rules section may be omitted.

Blanks, tabs and newlines are ignored, except that they may not appear in names or multi-character reserved symbols. Comments are enclosed in /\* ... \*/ and may appear wherever a name is legal.

Names may be of arbitrary length, made up of letters, dot ., underscore \_ , and non-initial digits. Upper and lower case letters are distinct. Names beginning in yy should be avoided, since the *yacc* parser uses such names.

A literal consists of a character enclosed in single quotes. The C escape sequences (e.g., '\n') are recognised.

### Declarations

The following declarators may be used in the declarations section:

#### %token

Names representing tokens must be declared; this is done by writing

```
%token name1 name2 ...
```

in the declarations section. Every name not defined in this section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one grammar rule.

#### %start

The *start symbol* represents the largest, most general structure described by the grammar rules. By default, it is the left hand side of the first grammar rule; this default may be overridden by declaring:

```
%start symbol
```

#### %left

#### %right

#### %nonassoc

Precedence and associativity rules attached to tokens are declared using these keywords. This is done by a series of lines, each beginning with



one of the keywords *%left*, *%right*, or *%nonassoc*, followed by a list of tokens. All tokens on the same line have the same precedence level and associativity; the lines are in order of increasing precedence or binding strength. *%left* denotes that the operators on that line are left associative, and *%right* similarly denotes right associative operators. *%nonassoc* denotes operators that may not associate with themselves. (A token declared using one of these keywords need not be declared by *%token* as well.)

#### **%prec**

Unary operators must, in general, be given a precedence. In cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences, the keyword *%prec* is used to change the precedence level associated with a particular grammar rule. *%prec* appears immediately after the body of the grammar rule, before the action or closing semicolon (see Grammar Rules below), and is followed by a token name or a literal. It causes the precedence of the grammar rule to become that of the following token name or literal.

#### **%union**

By default, the values returned by actions and the lexical analyser are integers. Other types, including structures, are supported: the *yacc* value stack is declared to be a union of the various types of values desired. *Yacc* keeps track of types, and inserts appropriate union member names so that the resulting parser will be strictly type-checked. The declaration is done by including a statement of the form:

```
%union {  
    body of union  
}
```

Alternatively, the union may be declared in a header file, and a typedef used to define the variable *YYSTYPE* to represent this union. The header file must be included in the declarations section by using a *#include* construct within *%{* and *%}* (see below). Union members must be associated with the various names. The construction *<name>* is used to indicate a union member name; if this follows one of the keywords *%token*, *%left*, *%right*, and *%nonassoc*, the union member name is associated with the tokens listed.

#### **%type**

This key word is used to associate union member names with nonterminals, in the form:

```
%type <ntype> a b ...
```

Other declarations and definitions can appear in the declarations section, enclosed by the marks `%{` and `%}`. These have global scope within the file, so that they may be used in the rules and programs sections.

#### Grammar Rules

The rules section is comprised of one or more grammar rules. A grammar rule has the form:

`A : BODY ;`

*A* represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are yacc punctuation. If there are several successive grammar rules with the same left hand side, the vertical bar `|` can be used to avoid rewriting the left hand side; in this case the semicolon must occur only after the last rule. The *BODY* part may be empty to indicate that the nonterminal symbol matches the empty string.

The ASCII NUL character (0 or `\0`) should not be used in grammar rules.

With each grammar rule, the user may associate actions to be performed each time the rule is recognised in the input process. These actions may return values and may obtain the values returned by previous actions. In addition, the lexical analyser can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input or output, call subprograms, and alter external variables. An action is one or more statements enclosed in braces `{` and `}`. Certain pseudo-variables can be used in the action: a value can be returned by assigning it to `$$`; the variables `$1`, `$2`, ... refer to the values returned by the components of the right side of a rule, reading from left to right. By default, the value of a rule is the value of the first element in it. Actions may occur in the middle of a rule as well as at the end; an action may access the values returned by symbols (and actions) to its left, and in turn the value it returns may be accessed by actions to its right.

Internal rules to resolve ambiguities are:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the grammar rule that occurs earlier in the input sequence.

In addition, the declared precedences and associativities (see Declarations Section above) are used to resolve parsing conflicts as follows:

1. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` keyword is used, it overrides this default. Some grammar rules may have no precedence and associativity.
2. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and



associativity, then the two rules given above are used.

3. If there is a shift/reduce conflict, and both the grammar rule and the input symbol have precedence and associativity associated with them, then the conflict is resolved in favour of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies error.

Conflicts resolved by precedence are not counted in the shift/reduce and reduce/reduce conflicts reported by *yacc*.

The token name *error* is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected and recovery might take place. When an error is encountered, the parser behaves as if the token *error* were the current lookahead token and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a series of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

The statement

```
yyerrok;
```

in an action resets the parser back to its normal mode; it may be used if it is desired to force the parser to believe that an error has been fully recovered from.

The statement

```
yyclearin;
```

in an action is used to clear the previous lookahead token; it may be used if a user-supplied routine is to be used to find the correct place to resume input.

### Programs

The programs section may include the definition of the lexical analyser *yylex* and any other functions; for example those used in the actions specified in the grammar rules.

*Yylex* is an integer-valued function which returns the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yyval*. The parser and *yylex* must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by *yacc* or chosen by the user. In either case, the *#define* construct of C is used to allow *yylex* to return these numbers symbolically. If the token numbers are chosen by *yacc*, literals are given the numerical value of the character in the local character set and other names are assigned token numbers starting at 257.



A token may be assigned a number by following its first appearance in the declarations section with a nonnegative integer. Names and literals not defined this way retain their default definition. All token numbers must be distinct.

The end of the input is marked by a special token called the *endmarker*. The endmarker must have token number 0 or negative. (These values are not legal for any other token.) All lexical analysers should return 0 or negative as a token number upon reaching the end of their input. If the token up to but excluding the endmarker forms a structure which matches the start symbol, the parser accepts the input. If the endmarker is seen in any other context, it is an error.

## FILES

|          |                           |
|----------|---------------------------|
| y.output | report file               |
| y.tab.c  | program generated by yacc |
| y.tab.h  | token code definitions    |

## ERRORS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the *y.output* file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

## SEE ALSO

lex(1D).

## CHANGE HISTORY

First released in Issue 2.

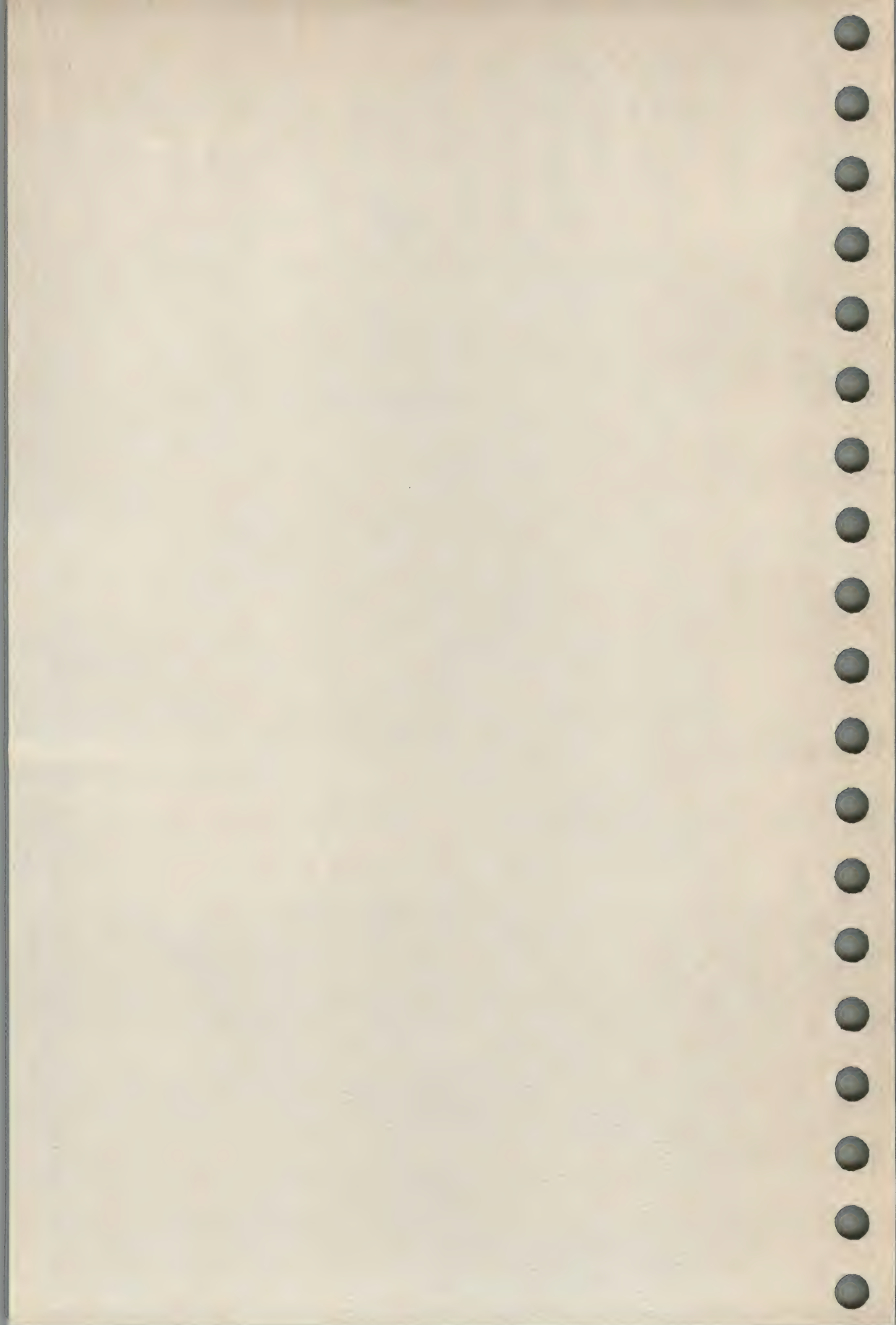
## Issue 2

Derived from the entry in Issue 2 of the SVID.

# X/O/P/E/N/

PORTABILITY GUIDE

THE X/OPEN SYSTEM V SPECIFICATION  
COMMANDS AND UTILITIES APPENDIX A

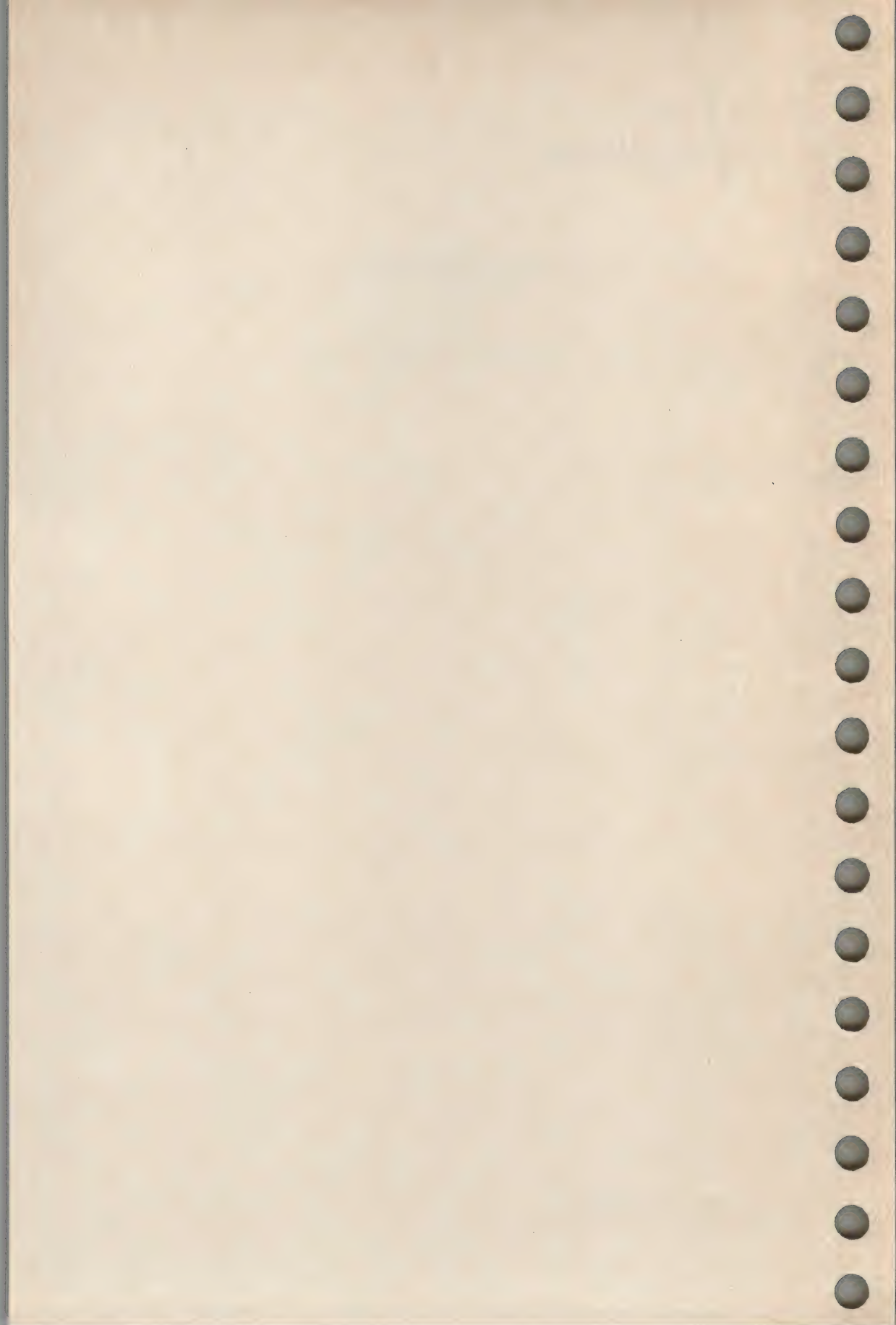






# Contents

|          |     |                             |
|----------|-----|-----------------------------|
| Appendix | A   | ENHANCED DEFINITIONS        |
|          | A.1 | INTRODUCTION                |
|          | A.2 | SPECIFICATION TEMPLATE      |
|          | A.3 | EXAMPLES OF IMPROVED FORMAT |
|          |     | <i>cat(1)</i>               |
|          |     | <i>rm(1)</i>                |





## Enhanced Definitions

### A.1 INTRODUCTION

There is a strong body of opinion that the general methods of defining Commands and Utilities in current usage are unsatisfactory, specifically being incomplete and displaying a lack of precision in their specifications.

This Appendix represents a first attempt to establish an improved method of definition and presentation. Two simple commands, *cat* and *rm*, have been selected to illustrate the technique and to test the format. In developing a future full set of definitions along these lines, a primary intent would be to establish a consistent format and to apply it rigorously.

In preparing the two examples contained in this Appendix no attempt has been made to improve the quality of specification of the information contained, and, in this sense, the examples directly reflect the corresponding definition pages in the main body of the XVS. Specifically, portability warnings such as PI and OF have been carried over directly.

Consequently, any entry marked as "UNDEFINED" in these examples represents a major area of future work for the X/OPEN Group in developing a complete set of Commands and Utilities defined to a consistent and high level of detail. In the interests of promoting common standards and practices it is the firm intention that this work will be undertaken in close collaboration with the appropriate standards bodies.

The process of creating these examples has forcefully reminded the X/OPEN members of the magnitude of the task, and the difficulty of fully specifying the semantics of commands where a large number of valid combinations of options exist, particularly when the results and consequences are by no means readily apparent from the information contained in current manual pages. The task appears deceptively simple, until you try to carry it out! (Note that in these examples comments on standard input and standard output have not been tested on the systems of all X/OPEN member companies.)

The preparation of material of this type requires a recognition of the possible conflict between creating user documentation and creating a specification. X/OPEN does not claim that these examples represent the definitive answer and readers' constructive comments are welcomed as input to future work on this task.

The remainder of this appendix comprises a full description of the "specification template" adopted for the two examples, followed by the two examples themselves.



## A.2 SPECIFICATION TEMPLATE

|             |                                                                                                                                                                                                                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NAME        | specifies the name of the command and gives a brief indication of its function.                                                                                                                                                                                                                 |
| SYNOPSIS    | defines the complete syntax for the command, its options and operands. (The notation used for the synopsis will be defined fully in the complete Commands and Utilities definition in a future Portability Guide.)                                                                              |
| DESCRIPTION | contains a brief description of the general operation of the command.                                                                                                                                                                                                                           |
| OPERANDS    | describes the use of <i>stdin</i> , <i>stdout</i> , and <i>stderr</i> by the command and also defines the manner in which additional operands, typically file-name lists, are interpreted. Where multiple operands are accepted, the consequences of any interaction between them is described. |
| OPTIONS     | lists in alphabetical order each valid command option, accompanied by a description of its modifying effect on the command. Options may be divided into sub-groups which are functionally related, e.g., those which affect output format may be listed in one group.                           |

Where a command accepts multiple options, the interaction between commonly used combinations is described.

## EXTERNAL INFLUENCES

defines those aspects of the command's executing environment that affect its operation. The names of all environmental variables that are used or checked by the command are specified. The names of any files whose presence is assumed, or whose presence is checked for, are also defined. Any modification of the command action based on the presence, absence or contents of files, is specified.

**INTERACTIONS** describes the commonly encountered interactions between combinations of operands, options and external influences.

## EXTERNAL EFFECTS

notes files explicitly created, written, or removed by the command. In addition signals sent, processes created, or, in the case of shell built-in commands, shell internal states that are changed are specified. Such information is provided to enable software writers and system users to avoid name conflicts etc. when running applications.



**EXTENDED DESCRIPTION**

fully describes those more complex commands, such as *make*, *awk*, which fall short of needing an entire programming language description but whose behaviour cannot be adequately described under the **DESCRIPTION** and **OPTIONS** headings.

**EXIT STATUS**

describes the command's exit status, if one is produced. All values of the exit status are detailed.

**ERRORS**

lists (to the extent feasible), all error messages that the command may produce with interpretations and corrective actions as appropriate. Those error conditions which are specifically checked for by the command are specified.

**EXTENSIONS**

highlights where existing implementations have already defined additional options, and where future directions are known. This section identifies ways in which implementers can avoid potential conflicts. If there are specific extension options reserved for implementers or application developers, these are described here.

**NOTES**

contains any additional information useful to the user of the command which is not included elsewhere.

Neither the **EXTENSIONS** or the **NOTES** section should properly be considered part of the standard.

References to external documents, to other parts of the same document, and portability considerations for comparable commands which are not "standard-compliant" are given here.

**FUTURE DIRECTIONS**

should be used as a guide to current thinking. If the command is known, or believed, to be subject to change in the future, then useful information pertaining to that, including the possible removal of the command, is presented.

**CHANGE HISTORY**

records changes made to the command description since its earliest publication in this format.



**A.3 EXAMPLES OF IMPROVED FORMAT**

Two simple examples are included to illustrate the use of improved command definition template.

A necessary prerequisite of any precise definition is a full definition of the syntax rules and of all terms used.

For the purposes of these examples:-

- a. Rules for command-line syntax as shown in GETOPT (SVID Issue 2 Volume 1, page 182) have been assumed and adopted.
- b. References to file-types are defined as:

```
File ::=      Regular File
              | <Directory>
              | <Character Special File>
              | <Block Special File>
              | <Fifo>
```

```
Regular File ::= Text File
                  | Binary File
```

```
Text File ::=   <Plain Text File>
                  | <Executable Text File>
```

```
Binary File ::= <Data File>
                  | <Executable File>
```

Definitions of "Directory", "Block Special File", "Character Special File" and "Fifo Special File" can be found in the introduction to "XVS COMMANDS AND UTILITIES". The terms "Plain Text File", "Executable Text File", "Data File" and "Executable File" are not yet fully defined but are intended to have an intuitive meaning.